Sistema de matchmaking para un videojuego multijugador

Por
José Martín Serrano, Pablo García Grossi, Javier Arias González, Ignacio Ory Alonso

UNIVERSIDAD COMPLUTENSE
MADRID

Grado en Desarrollo de Videojuegos
Facultad de Informática

Dirigido por
Pedro Pablo Gómez Martín, Guillermo Jiménez Díaz

Matchmaking system for a multiplayer video game

Madrid, 2020–2021

Sistema de matchmaking para un videojuego multijugador

Memoria que se presenta para el Trabajo de Fin de Grado

José Martín Serrano, Pablo García Grossi, Javier Arias González, Ignacio Ory Alonso

Dirigido por

Pedro Pablo Gómez Martín, Guillermo Jiménez Díaz

Departamento de Ingeniería del Software e Inteligencia Artificial
Facultad de Informática
Universidad Complutense de Madrid

Madrid, 2021

Agradecimientos

En primer lugar, queremos agradecer a todos nuestros familiares, amigos y a nuestra mascota, Peepy, por el constante apoyo y ánimo que nos han dado.

A Gonzalo y Guillermo, compañeros que nos han ayudado durante toda la carrera, pero aún más en el desarrollo de este proyecto.

A nuestros tutores, Guillermo Jimenez Díaz y Pedro Pablo Gómez Martín, por su labor, guiándonos o tirándonos de las orejas según fuera necesario.

No podía faltar un reconocimiento a toda la gente (imposible mencionarlos uno a uno) que participó en las pruebas con usuarios y difundió las mismas.

Resumen

Sistema de matchmaking para un videojuego multijugador

Uno de los problemas frecuentes en los videojuegos multijugador en red es la generación de partidas "equilibradas" donde todos los jugadores se diviertan y puedan disfrutar de un reto que se adecúe a su nivel de habilidad. Mientras que en los videojuegos *offline* el reto está estandarizado y es el mismo para todos los jugadores pues es establecido por los propios diseñadores, en un juego multijugador, lograr este objetivo se consigue al emparejar a jugadores con las mismas aptitudes. Los sistemas de *matchmaking* son los responsables de generar emparejamientos de jugadores teniendo en cuenta este concepto de equilibrio.

El objetivo de este trabajo es el diseño e implementación de un sistema de *matchmaking* que será aplicado a un videojuego de género *shooter* con multijugador en línea, de perspectiva 2D, que incluirá distintos personajes y armas. Este sistema se encargará de asignar un nivel de puntuación a cada jugador en relación a su habilidad y actualizarlo según su resultado en las partidas, además de asignar a cada jugador interesado en jugar una partida un rival que ronde su mismo nivel de habilidad para lograr una partida equilibrada para ambos jugadores.

Para mantener el nivel de puntuación correcto de cada jugador, el sistema recogerá información acerca de la actuación de los jugadores durante las partidas y lo empleará para actualizar su puntuación y establecer mediciones precisas de su habilidad en base a sus resultados.

Palabras clave

Videojuego, Videojuego Multijugador, Emparejamiento de jugadores, Matchmaking, Multijugador competitivo

Abstract

Matchmaking system for a multiplayer video game

One of the frequent problems in networked multiplayer video games is the generation of 'balanced' games where all players have fun and can enjoy a challenge that suits their skill level. While in offline games the challenge is standardised and is the same for every player as it is set by the designers themselves, in an online game achieving this goal is achieved by matching players with the same skills. The matchmaking systems are responsible for generating player pairings with this concept of balance in mind.

The aim of this project is the design and implementation of a matchmaking system which will be applied to a shooter videogame, which will feature online multiplayer, 2D perspective and different characters and weapons.

This system will be in charge of assigning a score level to each player based on their skill and updating it according to their results in the games, as well as assigning each player interested in playing a game an opponent with the same skill level in order to achieve a balanced game for both users.

In order to maintain the correct rating level for each player, the system will collect information about the player performance during games and will use it to update the rating and establish accurate measures of the skill based on the results.

Keywords

Videogame, Multiplayer videogame, Player pairing, Matchmaking, Competitive multiplayer

Índice general

Página

Parte I

Español

Capítulo 1

Introducción

1.1. Motivación

En los últimos años, el formato de videojuego multijugador en línea ha tenido un importante auge y muchos juegos se han adaptado para ofrecer esta experiencia de juego en la que los usuarios puedan medir sus habilidades contra otros jugadores. Para lograr la mejor experiencia y conseguir cautivar a los jugadores, a lo largo de los años se han ido diseñando, mejorando o reimplementando distintos sistemas de emparejamiento (creación de parejas) y clasificación (clasificación de jugadores) tanto en juegos tradicionales [1, 2, 3, 4] como en videojuegos multijugador en línea [5, 6, 7, 8, 9, 10, 11, 12].

Uno de los principales desafíos a la hora de lograr una competición que resulte agradable y divertida para los usuarios es crear sistemas de emparejamiento y clasificación que les permitan enfrentarse a usuarios de su nivel, ofreciéndoles así un reto adecuado para sus capacidades.

Debido a esto, se han realizado numerosos estudios (detallados en la sección 2 *Estado Del Arte*) y planteamientos de posibles sistemas de *matchmaking* (creación de parejas) asociado a los videojuegos multijugador, especialmente en aquellos con modos de juego competitivos. La mayoría de estos se han constituido bajo la premisa de que cada jugador posee una habilidad determinada que es representada por un valor numérico, o *rating*, y es asignada por un sistema de clasificación. Este valor es también conocido de forma coloquial como "Elo", nombre que referencia el sistema de clasificación Elo, originalmente desarrollado para ser aplicado al ajedrez y que fue implementándose en diferentes videojuegos como una forma sencilla de emparejar jugadores en línea. Poco a poco, se han ido implementando nuevos sistemas derivados de este método inicial usado para el emparejamiento en el ajedrez.

Estos sistemas tienen como objetivo establecer una medición de las aptitudes de los usuarios teniendo en cuenta diferentes factores (como resultados finales, comportamientos y movimientos realizados durante la competición, etcétera). Estos factores después permiten juntar aquellos jugadores cuyas cualidades sean mejores dentro del ámbito de un videojuego o deporte en concreto.

Existen diversas filosofías de diseño que ofrecen múltiples opciones de cara a desarrollar sistemas de emparejamiento, todos derivados de los originales del ajedrez. Muchos de ellos

usan el sistema clásico de Elo previamente mencionado, y lo amplían o modifican para obtener una medición más certera de la puntuación y un emparejamiento más fluido y preciso. El problema es que en muchos casos estas implementaciones son poco accesibles, bien sea por falta de código abierto que sea fácil de reutilizar, o bien por la propia complejidad de este campo. Así pues, lo que buscamos es crear nuestra propia implementación que incluya sus propios sistema de emparejamiento y un sistema de clasificación.

1.2. Objetivos

El objetivo del proyecto es diseñar y desarrollar un sistema de *matchmaking* aplicable a un juego multijugador en línea. El sistema funcionará mediante un servidor que tendrá acceso a una base de datos de usuarios y empleará la información de cada uno de ellos para decidir los emparejamientos. Aparte, deberá poder actualizar la puntuación de cada uno de ellos de acuerdo a su actuación en partidas previas.

Para diseñar el sistema de *matchmaking*, se realizará un estudio de los sistemas ya existentes y en uso para partir de una base estable. También se estudiarán métodos de clasificación de jugadores, un componente importante a la hora de emparejarlos.

De la misma manera, se elaborará un videojuego multijugador *online* para probar este sistema de emparejamiento, además de hacer las veces de caso de uso para todas las pruebas correspondientes. Se realizarán una serie de pruebas con jugadores usando este juego, con el fin de determinar el rendimiento y la calidad de nuestro sistema.

Como meta adicional, queremos diseñar este sistema de forma que pueda ser implementado y adaptado a cualquier tipo de videojuego en línea, de manera que pueda ser reutilizado en el futuro por cualquier usuario.

1.3. Plan de trabajo

Para el desarrollo de este trabajo se empleará a nivel interno, con ciertas libertades, la metodología ágil de desarrollo Scrum, dividiendo el trabajo a realizar en diferentes historias de usuario para repartirlas en función de la disponibilidad, la capacidad y el conocimiento en el área.

Para el seguimiento del progreso en el desarrollo tendremos reuniones semanales para ir adaptando nuestras tareas e ir estableciendo nuevas metas. Ya hemos empleado esta metodología de trabajo en múltiples asignaturas (como son Metodologías Ágiles de Producción, así como Proyecto en todos los años) con resultados satisfactorios para todas las partes, haciendo nosotros mismos de clientes de cara al resto del equipo.

Las reuniones con los tutores se utilizarán para establecer los diferentes niveles de prioridad para las próximas tareas a realizar, así como para resolver diferentes dudas y consultas de corte más técnico.

Primeramente, investigaremos diferentes sistemas de *matchmaking* empleados en otros videojuegos para establecerlos como ejemplo a la hora de implementar el nuestro. Posteriormente, se irá diseñando la arquitectura más básica de este sistema de *matchmaking*, teniendo en cuenta las restricciones que debamos aplicar según el foco del mismo. Y una vez diseñada, implementaremos esta arquitectura y realizaremos la conexión entre

clientes y servidor. Así mismo, probaremos la implementación en busca de errores, simulando conexiones desde clientes y observando que se realicen todos los pasos de forma correcta.

Por último, aunque estrictamente relacionada pero sin ser el foco principal del trabajo, se llevará a cabo el desarrollo del videojuego que emplearemos como caso de uso, para tener unas bases sobre las que probar el sistema de *matchmaking*. Realizaremos pruebas con usuarios reales empleando este caso de uso, dejando que jueguen de forma libre para estudiar el rendimiento del sistema y los datos recogidos de estos jugadores.

1.4. Estructura de la memoria

En el capítulo 2 *Estado Del Arte* se detalla la investigación de sistemas de *matchmaking* y las conclusiones extraídas, además de nuestras decisiones en cuanto a qué sistemas usar de referencia o implementar. El diseño de nuestro sistema se encuentra en el capítulo 3 *Diseño del sistema de matchmaking*. Por otro lado, el capítulo 4 *Implementación* detalla la implementación de este sistema, y su fase de pruebas.

El documento de diseño del juego que empleamos como caso de uso se encuentra en el capítulo 5 *Caso de estudio*, y las pruebas con usuarios en el capítulo 6 *Prueba con usuarios*.

URLs de los repositorios empleados para este trabajo:

- Repositorio general, con el resto incluidos como submódulos:
  - https://github.com/HoracioStudios/TFG
- Servidor de Matchmaking:
  - https://github.com/HoracioStudios/Matchmaking-Server
- Sistema de actualización de puntuaciones:
  - https://github.com/HoracioStudios/Ranking-Update
- Controlador de servidores de juego:
  - https://github.com/HoracioStudios/ControlServidoresTeFeGe
- Caso de uso:
  - https://github.com/HoracioStudios/TeFeGe
- Librería de comunicación entre juego y servidor:
  - https://github.com/HoracioStudios/ClientCommunication
- Conexión con la base de datos:
  - https://github.com/HoracioStudios/MongoJS

Capítulo 2

Estado Del Arte

El auge de los videojuegos multijugador en línea con modos competitivos ha generado interés en diseñar métodos para emparejar jugadores entre sí. Por esto mismo, nuestra investigación acerca de este tema comienza principalmente por el análisis de diferentes modelos ya existentes y en uso, estudiar sus características y beneficios y, sobre todo, entender su funcionamiento y cómo emplearlos correctamente.

Así pues, en este capítulo detallaremos los resultados de nuestro proceso de investigación: hablaremos del concepto de *matchmaking* y los tipos que hay, los distintos sistemas que hemos encontrado y que ya se emplean en diversos ámbitos (tanto competiciones físicas como videojuegos en línea) y concluiremos con nuestra decisión final acerca de qué modelo o sistema emplearemos en nuestro trabajo.

2.1. *Matchmaking*: Qué es y qué formatos existen

El *matchmaking*, o emparejamiento en español, es un proceso por el cual se agrupan jugadores con habilidad similar en juegos competitivos empleando un sistema de clasificación que determine de forma numérica esta habilidad. En el contexto de videojuegos multijugador en línea, se necesita además una infraestructura que se encargue de recibir peticiones de jugadores y emparejarlos entre sí según estas clasificaciones. De esta manera, se pueden definir dos componentes en todo este proceso: el sistema de clasificación, que determinará estos valores numéricos (puntuaciones) que reflejen la habilidad de un jugador, y el sistema de *matchmaking*, que emparejará jugadores según estos valores.

Para diseñar todo este proceso, primero se debe empezar con la búsqueda de información acerca de la implementación del emparejamiento en varios contextos, sean videojuegos u otros deportes clásicos que utilicen sistemas similares.

El ajedrez fue el ámbito en el que comenzaron este tipo de investigaciones. Al ser un juego muy antiguo, las federaciones se formaron mucho antes que en otros casos, así como las competiciones. Con ellas, surgió la necesidad de emparejar jugadores según su nivel de habilidad. Algunos métodos empleados para determinar este aspecto incluyen el Sistema de Harkness [1], el Sistema de la Federación Inglesa de Ajedrez [4] o, el estándar hoy en día, el Sistema Elo [3].

Comenzando por los sistemas de clasificación, estos se basan en asignar un valor numérico

por defecto a todo jugador recién llegado. Este valor incrementará o disminuirá en función del resultado de las partidas jugadas posteriormente y el valor numérico de los oponentes, representando así su habilidad. Este cálculo también puede incluir más factores y ser así más complejo, pero todos los sistemas comparten esta base.

Empleando estas clasificaciones se permitirá a los jugadores enfrentarse entre sí en función de los emparejamientos que se produzcan y según el sistema de clasificación, bien tras cada partida o bien tras un periodo de tiempo, se actualizará la valoración de estos jugadores según los resultados y el sistema.

Los sistemas de emparejamiento emplean el nivel de los jugadores asignado por los sistemas de clasificación para determinar los enfrentamientos, y debe refinarse con el objetivo de que las partidas sean satisfactorias para ambas partes. Así pues, ambos sistemas están estrechamente relacionados. La forma de ajustar las puntuaciones de los jugadores debe ser coherente para que realmente mida la habilidad de cada usuario, y así ser útil para emparejamientos futuros. De la misma manera, la forma en la que se realicen los emparejamientos debe ser razonable en función de las puntuaciones, y "predecir" cómo de desequilibradas serán las partidas.

Según su enfoque, podemos definir dos tipos generales de sistemas de emparejamiento:

- **Sistemas que recomiendan partidas**. Estos primero crean una partida para un jugador, y se le asigna un nivel de habilidad medio. Después, se asigna esta partida a otros jugadores cuyo nivel sea similar, y se vuelve a calcular el nivel medio de la partida. Este sistema es adecuado para videojuegos multijugador en los que participan más de 2 jugadores.
- **Sistemas que recomiendan jugadores**. Estos se centran en la búsqueda de jugadores de un nivel similar al del usuario que desea jugar, pudiéndose adaptar a sistemas de emparejamiento tanto para partidas con dos jugadores, como aquellas con un número mayor.

La diferencia radica en el hecho de que el primer tipo solo se aplica a partidas dinámicas, en las cuales se puedan incorporar y salir jugadores durante su transcurso, mientras que el segundo tipo se aplica a partidas estáticas, que solo comiencen al encontrarse suficientes jugadores y no permitan que se incorporen más. También existen sistemas híbridos, que aplican el segundo método pero de desconectarse un jugador, permiten que se incorpore otro empleando el primer método.

En la sección siguiente se detallan algunos de los sistemas de clasificaciones más habituales, tanto en videojuegos como en juegos clásicos.

2.2. Sistemas de clasificación

A la hora de implementar un sistema de emparejamiento, el primer paso es seleccionar el sistema de calificación. No existe uno perfecto, ya que cada uno ofrece cualidades distintas: simplicidad, precisión, flexibilidad, etcétera. Así pues, a la hora de elegirlo conviene primero definir qué requisitos se tienen. El principal requisito que vamos a aplicar será tener de partidas de 2 jugadores, uno contra uno.

A continuación se detallan los 4 sistemas que consideramos más aplicables a nuestro proyecto.

2.2.1. Elo

El sistema de Elo fue inventado por Arpad Elo [3] en la década de los 60, y surgió para crear un sistema que permitiera medir la habilidad de los jugadores de ajedrez, para así poder emparejarlos y clasificarlos según dicha puntuación, la cual podrían adquirir participando en competiciones.

Este modelo se mantiene hoy en día en los torneos de ajedrez y es usado para los emparejamientos en las primeras fases de la competición, evitando emparejar a los jugadores con mayores puntuaciones en las rondas iniciales, ya que estos son los que más posibilidades tienen de ganar el torneo. La clasificación por puntuación de los jugadores también se utiliza para restringir el acceso a ciertos torneos y poder crear competiciones de jugadores de un nivel similar.

El rango de puntuación se encuentra comprendido entre 0 puntos y 3000 puntos, y varía a lo largo del tiempo según los resultados de las partidas de los torneos tenidos en cuenta para el cálculo de la misma.

Elo se fundamenta en la idea de que cada jugador posee una habilidad determinada por un valor numérico, y este valor se actualiza tras cada partida teniendo en cuenta el resultado de esta, además de una predicción previa del resultado.

Para este proceso, el modelo usa un sistema de predicción (ecuación 2.1) que permite identificar aquel jugador con más probabilidades de victoria en la partida.

La probabilidad de victoria E_a de un jugador a se calcula con la fórmula 2.1, que describe una función sigmoide como aparece en la figura 2.1

$$E_a = \frac{1}{1 + 10^{-(R_a - R_b)/400}} \tag{2.1}$$

Donde:

- R_a representa la puntuación del jugador a.
- R_b representa la puntuación del jugador b.

El cálculo de la probabilidad de victoria del jugador b se calcula de forma equivalente.

Elo solo contempla 3 posibles resultados a la hora de dar por finalizada una partida: victoria, empate y derrota, cada una de ellas representada por los valores 1, 0,5 y 0, respectivamente.

Para la actualización de las puntuaciones de los usuarios, el sistema emplea las predicciones realizadas y los resultados de cada partida para hallar el nuevo valor de medición de la habilidad de los jugadores (ecuación 2.2). Se emplea la predicción para tener en cuenta la diferencia de habilidad entre jugadores: si el jugador con más probabilidad de ganar se hace con la victoria, obtendrá una cantidad de puntos menor que en el caso de hallarse en desventaja, y viceversa.

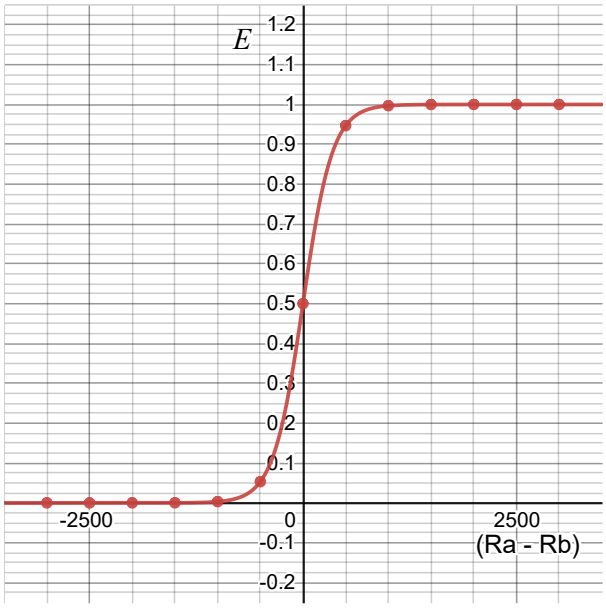

Figura 2.1: Función sigmoide descrita por la fórmula 2.1 El eje y representa la predicción entre 0 y 1, y el eje x representa la diferencia de puntuaciones entre los jugadores a y b

Para calcular la puntuación nueva del jugador a, representada como R'_a, se utiliza la fórmula:

$$R'_a = R_a + K(S_a - E_a) \tag{2.2}$$

Donde:

- R_a representa la puntuación previa a las partidas.
- K, conocido como "K-factor", es una constante que controla el máximo ajuste que puede tener una partida. En el ámbito del ajedrez, se suele emplear un valor de $K = 16$ para maestros y $K = 32$ para principiantes. Cuanto mayor sea el valor, mayor será el cambio de puntuación tras una actualización. Se opta por un valor superior para principiantes al no conocerse su valor real.
- S_a representa el resultados de la partida.
- E_a representa la probabilidad de victoria de la partida, que fueron estimadas previamente.

Elo es un sistema totalmente establecido y fácilmente aplicable a videojuegos en línea pero se encuentra actualmente en desuso en este ámbito. Esto se debe a que existen sistemas para realizar emparejamientos individuales que amplían o mejoran Elo, o que se ajustan mejor al ámbito de los videojuegos, además de que es un sistema difícilmente aplicable a juegos que involucren más contrincantes.

De esta manera, el sistema Elo queda principalmente relegado a simuladores de ajedrez, como Lichess [13], aunque estos suelen complementarlo con otros sistemas. Un ejemplo de este avance es que títulos como *League of Legends* (durante sus inicios, hasta el año 2013, pasando a un sistema propio que mantiene en la actualidad [7]) u *Overwatch* em-

plearon originalmente sistemas derivados de Elo, para pasar a sistemas propios en la actualidad.

Elo ha servido como punto de partida para muchos otros sistemas de clasificación, muchos de los cuales lo emplean como base.

2.2.2. Glicko

El sistema Glicko fue inventado por Mark E. Glickman [14] de la universidad de Boston para tratar de mejorar el sistema Elo.

La principal deficiencia que Glickman había detectado era que el sistema Elo no era lo suficientemente preciso a la hora de medir las habilidades de un jugador. Por ejemplo, habría cierta imprecisión en el caso de que un jugador llevase tiempo sin jugar ya que su habilidad podría haber cambiado, y por tanto su puntuación no reflejaría este cambio.

Para compensar este problema de fiabilidad, Glicko implementa el concepto de *ratings deviation* (RD), o desviación de puntuación en español, que mide la incertidumbre del valor de habilidad asignado a un usuario. Esta desviación será más alta cuanto más tiempo lleve un jugador sin competir o, en el caso de jugadores nuevos cuanto menor sea el número de partidas en las que haya participado. Glicko recomienda que a los jugadores nuevos se les asigne como valores por defecto una puntuación de 1500 y la desviación máxima posible, que recomienda como 350.

En el sistema Glicko tanto la puntuación como la desviación se actualizan según el resultado de una partida, pero la desviación de puntuación se modifica además según el periodo de tiempo que el jugador lleve sin registrar una partida.

Para llevar a cabo el cálculo de puntuación, el sistema Glicko agrupa las partidas en un periodo de puntuación y actualiza la puntuación y desviación de todos los jugadores de forma simultánea. Un periodo de puntuación puede ser un lapso de tiempo de duración variable, pudiendo durar desde un minuto hasta varios meses. Glicko tiene un funcionamiento más preciso cuando el número de partidas entre cada periodo de tiempo es moderado, y se recomienda una media de 5-10 partidas por periodo.

El cálculo de la nueva puntuación se realiza en tres pasos, cálculo temporal de desviación, cálculo de puntuación y cálculo de desviación final, los cuales se describen a continuación.

Cálculo temporal de desviación

Primero se realiza un cálculo inicial de la desviación de puntuación (RD), en la cual se sobrestima para tener en cuenta una mayor incertidumbre desde la última actualización (ecuación 2.3):

$$RD = \text{mín}\left(\sqrt{{RD_0}^2 + c^2 t}, 350\right) \tag{2.3}$$

- *350* es el valor de desviación de puntuación inicial asignado por defecto a un nuevo jugador.

- RD_0 representa la desviación de puntuación previa al cálculo.
- t representa la cantidad de periodos que el jugador lleva sin ser actualizado.
- c representa una constante que marca el crecimiento de la incertidumbre entre periodos de puntuación. El valor más habitual es 34,6, como se describe a continuación.

El valor de la constante c se calcula mediante la ecuación 2.4):

$$c = \sqrt{\frac{(350^2 - 50^2)}{100}} \approx 34{,}6 \tag{2.4}$$

Donde 350 y 50 representan la desviación inicial y la desviación asumida como la media, respectivamente, mientras que 100 representa la cantidad de periodos asumidos como necesarios para regresar desde una desviación de 50 a la inicial. Todos estos valores son los que Glicko ofrece por defecto, y de desearse cambiarlos tendrían que reflejarse los cambios en esta fórmula.

Cálculo de puntuación

Dadas m partidas, se calcula la nueva puntuación, r, con las siguientes fórmulas:

$$r = r_0 + \frac{q}{\frac{1}{RD^2} + \frac{1}{d^2}} \sum_{i=1}^{m} g(RD_i)(s_i - E(s|r_0, r_i, RD_i)) \tag{2.5}$$

Donde:

$$g(RD_i) = \frac{1}{\sqrt{1 + \frac{3q^2(RD_i^2)}{\pi^2}}} \tag{2.6}$$

$$E(s|r_0, r_i, RD_i) = \frac{1}{1 + 10^{\left(\frac{g(RD_i)(r_0 - r_i)}{-400}\right)}} \tag{2.7}$$

$$q = \frac{\ln(10)}{400} = 0{,}00575646273 \tag{2.8}$$

$$d^2 = \frac{1}{q^2 \sum_{i=1}^{m} (g(RD_i))^2 E(s|r_0, r_i, RD_i)(1 - E(s|r_0, r_i, RD_i))} \tag{2.9}$$

Donde:

- r_0 representa la puntuación original.
- r_i representa las puntuaciones de cada oponente.
- s_i representa el resultado de cada partida. Por defecto, 1 es una victoria, 0.5 es un empate y 0 es una derrota.

Cálculo de desviación final

Una vez hallada la puntuación nueva, se puede arreglar el sobre-ajuste realizado previamente en la desviación:

$$RD' = \sqrt{\left(\frac{1}{RD^2} + \frac{1}{d^2}\right)^{-1}} \tag{2.10}$$

Como cada jugador posee una puntuación y una desviación (RD), el valor de la habilidad demostrada por el usuario se indica en forma de intervalo, utilizando un intervalo de confianza del 95 %: $r \pm RD \times 2$.

Glicko ha tenido un uso bastante extendido en diferentes videojuegos. Pero como veremos a continuación, pocos siguen utilizándolo en favor de su versión posterior. De esta manera, Team Fortress 2 [12] queda como ejemplo principal de uso en la actualidad junto a otros proyectos sin ánimo de lucro como Pokemon Showdown [11], que lo combina con otros sistemas como el ya mencionado Elo.

2.2.3. Glicko 2

Glicko 2 [2] es una versión extendida de Glicko. Además de contemplar los parámetros de puntuación y desviación de puntuación de un usuario, incluye también la volatilidad (σ), que mide la oscilación esperada de la puntuación de un jugador. Este valor es mayor cuanto más errática sea la forma de jugar del usuario y varía mucho con respecto a la conducta esperada para un jugador de su puntuación.

Al igual que ocurre con Glicko y la desviación de puntuación, la volatilidad es utilizada para resumir la habilidad del jugador a través de un intervalo, y no con un valor en concreto.

De la misma manera que hace su predecesor, Glicko2 almacena los resultados de las partidas durante un "periodo de puntuación", tras el cual se calculan de forma simultánea las puntuaciones, desviaciones y volatilidades de todos los usuarios.

Este sistema funciona mejor para videojuegos en los que el número de partidas por periodo de puntuación tienda a ser relativamente alto, aunque la longitud del periodo es decidida por los propios desarrolladores.

La escala de puntuación de Glicko 2 es diferente a la utilizada por la versión original. Sin embargo, los pasos realizados asumen que los datos de entrada vienen bajo la escala original de Glicko y, durante el proceso, estos son adaptados a Glicko 2 y transformados de nuevo al final de todos los cálculos a la escala original de Glicko.

En esencia, se comienza por "traducir" los valores originales de Glicko a valores de Glicko 2, siendo μ equivalente a la puntuación (ecuación 2.11) y φ equivalente a la desviación de puntuación (ecuación 2.12). Como Glicko, Glicko 2 asume unos valores por defecto de 1500 para la puntuación, y 350 para la desviación.

$$\mu = (r - 1500)/173{,}7178 \tag{2.11}$$

$$\varphi = RD/173{,}7178 \tag{2.12}$$

Después, para el cálculo de la volatilidad, es necesario llevar a cabo dos pasos:

- Calcular v con la fórmula 2.13, la varianza estimada de la puntuación del usuario, únicamente basado en los resultados de las partidas.

$$v = [\sum_{j=1}^{m} g(\phi_j)^2 E(\mu, \mu_j, \phi_j)\{1 - E(\mu, \mu_j, \phi_j)\}]^{-1} \tag{2.13}$$

- Calcular la mejora de puntuación estimada en comparación con el periodo anterior con la fórmula 2.14, basándose únicamente en los resultados de partidas anteriores.

$$\Delta = v \sum_{j=1}^{m} g(\phi_j)\{s_j - E(\mu, \mu_j, \phi_j)\} \tag{2.14}$$

Los componentes principales de estas fórmulas son:

- s, los resultados de las partidas.
- $g(\phi_j)$, análoga a la fórmula 2.6, descrita en la fórmula 2.15.

$$g(\phi) = \frac{1}{\sqrt{1 + 3\phi^2/\pi^2}} \tag{2.15}$$

- $E(\mu, \mu_j, \phi_j)$, análoga a la fórmula 2.7, que permite realizar una predicción del resultado de una partida entre dos jugadores. Descrita en la fórmula 2.16.

$$E(\mu, \mu_j, \phi_j) = \frac{1}{1 + exp(-g(\phi_j)(\mu - \mu_j))} \tag{2.16}$$

A continuación se debe calcular, con los datos obtenidos previamente, el valor de la nueva volatilidad en el nuevo periodo de puntuación. Este proceso requiere varias iteraciones y fórmulas, y es más complejo que los pasos anteriores.

Tras haber calculado la nueva volatilidad, esta se emplea para calcular los nuevos valores de desviación de puntuación y de habilidad del usuario y, por último, se vuelven a transformar el RD y la puntuación a la escala original de Glicko.

En el caso de que el jugador no haya competido durante el periodo de puntuación, únicamente se calcula el valor de la desviación de puntuación, pues la volatilidad y la puntuación permanecen estables.

Como hemos dicho anteriormente, este sistema ha sido adoptado por gran cantidad de proyectos de varios géneros, como pueden ser Counter Strike: Global Offensive [6], Guild Wars 2 [9], Splatoon 2 [8] o Tetr.io [10].

2.2.4. TrueSkill

TrueSkill [15] está orientado de forma específica a videojuegos *online* por equipos (de más de un jugador en cada uno). Fue desarrollado por Microsoft Research y se utiliza en la plataforma Xbox Live. Este sistema busca extender Elo basándose en un algoritmo de inferencia Bayesiana, buscando solucionar los siguientes problemas que surgen al aplicarlo con videojuegos *online*:

- Los resultados de las partidas corresponden a un equipo de jugadores. Sin embargo, debe existir un único valor de puntuación para cada jugador para futuros emparejamientos, ya que los equipos nunca serán los mismos.
- Existe un ganador y un perdedor, pero los resultados no han de estar definidos únicamente por valores de victoria y derrota, puesto que hay jugadores que pueden haber aportado más a la partida independientemente del resultado final.

De esta forma, TrueSkill está orientado al emparejamiento en juegos *online* con equipos, independientemente del número de integrantes, y de como se enfrenten entre sí (un equipo contra otro, o todos contra todos), aunque también puede ser aplicado a juegos que enfrentan jugadores de manera individual.

En este sistema se mide la habilidad del jugador con una distribución Gaussiana N caracterizada por los valores μ (la "habilidad percibida", o estimación media de la habilidad del jugador) y σ (la confianza que tiene el sistema en el valor μ). De esta manera, se puede interpretar que $N(x)$ es la probabilidad de que la habilidad "real" (true skill) de un jugador sea x.

Aunque existen implementaciones abiertas en diferentes lenguajes de programación, TrueSkill está patentado por Microsoft y es de uso exclusivo para proyectos comerciales que obtienen la licencia correspondiente. Así pues, la implementación original es en su mayoría un secreto. Existe una versión revisada, llamada TrueSkill 2 [16], que sigue los mismos principios, puliendo y mejorando su funcionamiento. Los títulos que originalmente utilizaban TrueSkill pasaron a utilizar esta nueva versión, como es el caso de Halo 5 [5], creado por los equipos de desarrollo de Microsoft.

Existe documentación acerca de la implementación concreta del sistema en este juego, principalmente en forma de charlas [17]. Sin embargo, la implementación de TrueSkill como base derivaría en un proceso demasiado complejo como para justificarlo para este trabajo.

2.3. Selección de sistema de calificación: Glicko

El sistema de *matchmaking* diseñado debe adaptarse a nuestro requisito principal: gestionar partidas uno contra uno, cortas y frenéticas, en el que se puedan jugar muchas partidas en poco tiempo. Si bien el plan es que nuestro sistema de emparejamiento permita cambiar el sistema de clasificación de forma relativamente sencilla, es necesario decidir uno para este proyecto.

Es por esto que, tras estudiar estas opciones, optamos por el sistema de Glicko. Este sistema resulta atractivo ya no solo para nuestros requisitos principales, sino además a la hora de implementar un sistema fácilmente ampliable (especialmente con Glicko 2, al

estar diseñado con este objetivo).

Respecto al resto de opciones, en primer lugar, TrueSkill es propiedad de Microsoft y requiere una licencia de uso (además de centrarse en modos multijugador). Esto hace que se aleje de nuestros objetivos y quede por lo tanto descartado.

Por otro lado, Elo es un sistema muy documentado y de fácil comprensión e implementación, además de ser bastante extensible [18]. Sin embargo, es un sistema que, a pesar de ser totalmente funcional, se encuentra un poco desactualizado para su aplicación en videojuegos en la actualidad. Esto se debe a la creciente complejidad de estos, y a la tendencia de los usuarios a jugar de una forma más esporádica que en juegos como el ajedrez, lo cual introduce un elemento de incertidumbre a la hora de juzgar la habilidad de los mismos.

Por tanto, Glicko proporciona las siguientes ventajas a la hora de comenzar a desarrollar el sistema:

- Existe gran cantidad de documentación acerca del sistema, y su funcionamiento se basa en la aplicación de fórmulas matemáticas relativamente sencillas y rápidas que permiten un veloz cálculo y actualización de la puntuación de los usuarios.
- Al ser una ampliación del sistema Elo, ofrece una mayor fiabilidad a la hora de determinar las puntuaciones de los jugadores, aunque en consecuencia su funcionamiento sea más complejo
- La actualización a Glicko 2 es sencilla, los cambios que suponen aplicar esta extensión solamente implican la modificación de los cálculos a realizar y la implementación de la volatilidad. Este sistema permite una mayor precisión a la hora de buscar usuarios adecuados para el emparejamiento, aunque el proceso iterativo requerido para su cálculo es algo más lento que el de la versión original.
- Es un sistema genérico que ya ha sido utilizado en el sector por proyectos con propuestas diferentes. Esto incluye tanto juegos basados en enfrentamientos por equipos (como Team Fortress o Splatoon) como aquellos basados en duelos individuales (como Pokémon Showdown). Este hecho permite comprobar que se puede adaptar fácilmente a cualquier tipo de proyecto, y permite extensiones para el cálculo de resultados en las partidas.

La decisión de no emplear desde un principio Glicko 2 se debe a la simplicidad del juego y la naturaleza del trabajo. Probar que implementación de una volatilidad de puntuación es correcta o útil requeriría una gran base de jugadores jugando durante un periodo de tiempo muy amplio. Esto les permitiría aprender las mecánicas y establecer su propio estilo de juego, estableciendo así una conducta esperada por su parte. Pero esto es algo que no tenemos garantizado en ningún caso, no solo ya por el hecho de conseguir que jueguen más allá del tiempo necesario para aprender las mecánicas y establecer un cierto nivel de habilidad en una única sesión, sino que vuelvan al juego de manera regular.

Capítulo 3

Diseño del sistema de *matchmaking*

Decidido nuestro sistema de clasificación, podemos proceder a la funcionalidad del sistema de *matchmaking* en sí mismo. En este capítulo detallaremos cómo se ha diseñado este sistema, teniendo en cuenta la restricción que hemos impuesto de partidas 1 vs. 1 y el sistema de clasificación que hemos escogido, Glicko.

3.1. Diseñando un sistema de *matchmaking*

Como ya se comentó en la sección 2.3 *Selección de sistema de calificación: Glicko*, nuestra principal restricción es que nuestro juego va a enfrentar dos jugadores entre sí (o, lo que es lo mismo, uno contra uno). Así pues, de los dos tipos generales definidos en la sección 2.1 *Matchmaking: Qué es y qué formatos existen* escogeremos el segundo: nuestro sistema recomendará jugadores, ya que de esta manera la partida empieza cuando se hayan emparejado, y siempre tiene un punto de fin porque un jugador derrota a otro.

Para un sistema de este estilo se requiere una lista de espera en la cual anotar aquellos jugadores en busca de partida. Esta lista se empleará para realizar el proceso de *matchmaking* de forma individual: los jugadores realizarán peticiones, se tratará de buscar un buen candidato, y una vez dos jugadores se hayan emparejado mutuamente se les enviará a una partida en línea generada por el juego que se esté empleando. Una vez acabada la partida, los jugadores enviarán los resultados (y cualquier otro dato que se desee) a un historial donde se almacenarán hasta calcular sus nuevas puntuaciones. La figura 3.1 esquematiza este proceso.

Al margen del punto de vista técnico, existe cierta flexibilidad a nivel de diseño a la hora de implementar el sistema de *matchmaking*, con algunas de las decisiones tomadas detalladas más a fondo en la sección 3.4 *Cálculo de la habilidad*. Tomaremos como referencia diferentes publicaciones por parte de veteranos de la industria del videojuego sobre la decisión de tener en cuenta aspectos más allá de la victoria o derrota en la partida, tema que se trata en esta charla [20].

También teníamos que ser conscientes de nuestro alcance, y sabiendo que un sistema de *matchmaking* requiere una gran cantidad de jugadores para su correcto funcionamiento

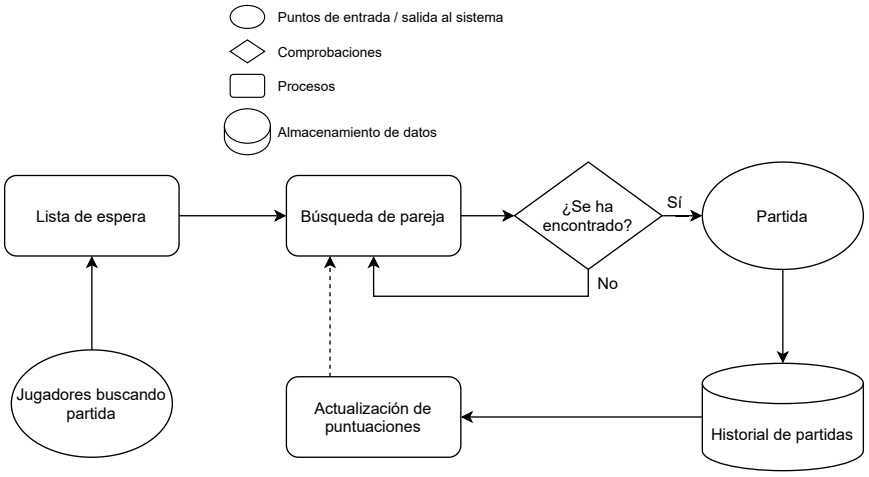

Figura 3.1: Esquema de la arquitectura de un sistema de *matchmaking*, basado en el descrito en *"Data Analytics Applications in Gaming and Entertainment"*[19]

[21], teníamos que tener en cuenta ciertas consideraciones a la hora de diseñar tanto el sistema como el juego a emplear como caso de prueba, para evitar todas las complicaciones posibles [22]. Por último, teníamos que tener en cuenta el efecto que puede llegar a tener el *matchmaking* sobre el estado del jugador [23] en casos extremos: por ejemplo, tener poca gente en la lista de espera puede derivar en tiempos de espera infinitos en caso de optar por un emparejamiento lo más igualado posible, o bien en partidas muy desigualadas en caso de optar por minimizar los tiempos para encontrar partida.

3.2. Proceso de *matchmaking*

En sí mismo, el proceso de *matchmaking* es relativamente simple. Funciona realizando peticiones a un servidor asíncrono, lo que significa que cuando un jugador hace una petición su conexión no se deja en espera, sino que se le contesta inmediatamente con la información disponible y solo se actualiza de recibirse una nueva petición.

Se puede ver desde dos puntos de vista. Desde la perspectiva del jugador, el proceso se compone de los siguientes pasos (figura 3.2):

1. El jugador realiza una petición de búsqueda de rival, enviando su identificador de usuario y el tiempo que lleva esperando.
2. Se recibe una respuesta a la petición.
3. Si la petición informa de que no se ha encontrado rival, se espera un tiempo, se aumenta el tiempo de espera y se vuelve al paso 1.
4. Si se ha encontrado rival, pueden darse dos casos:
    a) La petición informa de que el rival no ha sido emparejado de vuelta, o lo que es lo mismo, que solo ha habido un emparejamiento parcial. En este caso, se espera un tiempo y vuelve al paso 1 sin aumentar el tiempo de espera. Este paso se realiza para verificar que el emparejamiento es válido.
    b) La petición informa de que el rival ha sido emparejado, o lo que es lo mismo,

que se han emparejado mutuamente. En este caso se realiza una petición para ser eliminado de la lista de espera, y se procede a la sala de partida (o lo que se desee, según la implementación del juego). Este paso se realiza en el cliente para asegurar que ambos jugadores han avanzado a la partida correctamente antes de eliminarlos de esta lista.

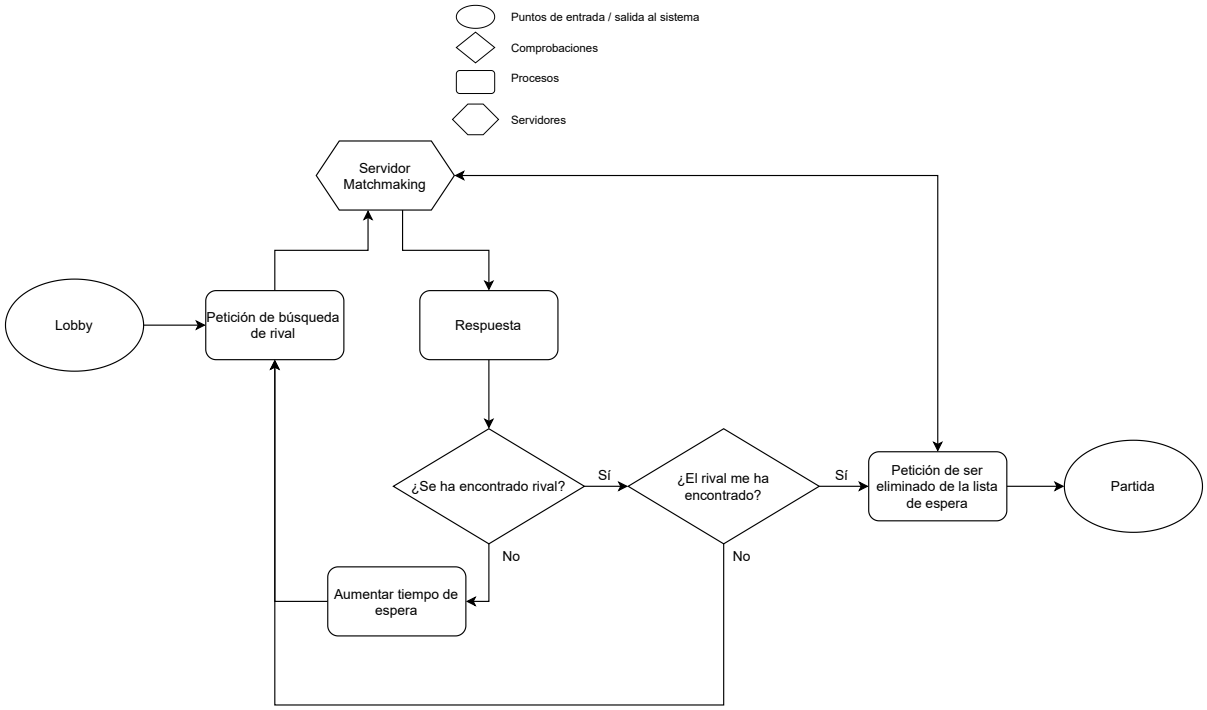

Figura 3.2: Esquema del proceso de *matchmaking* visto por el cliente. Se realizan dos comprobaciones: si le ha encontrado rival y si el rival ha sido emparejado con él. Las flechas bidireccionales representan respuestas que no necesitan procesado.

Y desde el punto de vista del servidor, como se detalla en la figura 3.3, el proceso consiste en:

1. El servidor recibe una petición de emparejamiento con un identificador de usuario y un tiempo de espera.
2. Si el identificador de usuario no está en la lista de espera, se añade. Si está, se actualiza la información y el marcador temporal (*timestamp*) de última petición, con tal de que no se elimine de la lista de espera por inacción (paso 3).
3. Se revisa la lista de espera, y todos los usuarios que lleven sin realizar una petición más de X segundos (por defecto se emplearán 5 segundos) se eliminan de la lista.
4. Se comprueba en la lista de espera si el identificador de usuario ya tiene asignado un rival. Este paso debe realizarse ya que el sistema es asíncrono.
    - Si no lo tiene, o lo tiene pero ya no está en la lista de espera, se realiza el emparejamiento según el método que se desee emplear. El tiempo de espera debería emplearse para ampliar el rango de búsqueda.
5. Cuando se encuentra un oponente, se le asigna el usuario como rival. Esto sirve para que cuando el oponente haga una petición, pueda saber que ya ha sido emparejado

parcialmente con otro usuario. En caso de que el usuario tuviera además asignado a dicho oponente, se considera que se han emparejado mutuamente.

6. Se envía una respuesta avisando de si se ha encontrado un rival, y de si están emparejados mutuamente.

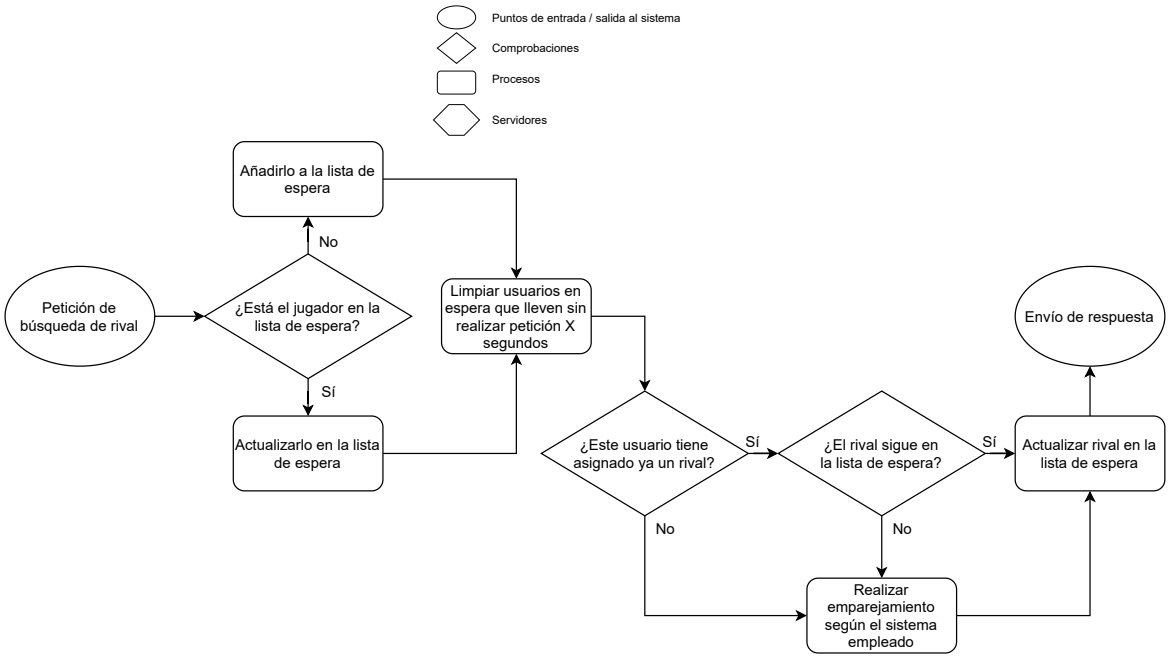

Figura 3.3: Esquema del proceso de *matchmaking* visto por el servidor

3.3. Actualización de puntuaciones

La actualización de puntuaciones depende del sistema de clasificación no solo en los cálculos, sino también a la hora de decidir cuándo realizarla. En el caso de Glicko, se recomienda actualizar las puntuaciones cada 5 días, teniendo en cuenta las partidas jugadas durante dicho periodo y simulando que todas han tenido lugar en el mismo instante. Esta cantidad es el resultado de calcular el tiempo necesario para realizar 10 partidas completas, con cierto tiempo añadido. Así pues, este valor debería modificarse según el juego y la duración media de sus partidas, según deseen los desarrolladores.

Las partidas pendientes no se eliminan, sino que se guardan en un historial. Así se permite revisar este historial, además de permitir la opción de cambiar o ajustar el sistema de clasificación, calculando las puntuaciones en base a este historial.

3.4. Cálculo de la habilidad

Para realizar los cálculos de la puntuación, se necesitan los resultados de las partidas jugadas. Con el fin de generalizar el formato de partidas, se considerarán siempre como subdivididas en rondas. Se tratarán los resultados de cada una de estas rondas de forma individual con tal de mantener la estructura de resultados señalada en la documentación de Glicko, en la que victoria equivale a 1, empate a 0,5 y derrota a 0. En caso de no realizar partidas divididas por rondas, se podrá considerar que la partida solo ha tenido

una ronda. Estos valores después se introducirán en las fórmulas descritas en la sección 2.2.2 *Cálculo de puntuación*.

A los jugadores nuevos se les aplicarán los valores por defecto recomendados por Glicko (sección 2.2.2): puntuación de 1500 y la desviación máxima de 350.

3.4.1. Consideraciones

Aparte de los parámetros, hay también una serie de casos especiales a considerar, en caso de que una partida no se pueda completar:

- El jugador se desconecta: Con tal de castigar al jugador en caso de desconexión intencionada, se contarán tres rondas como V = 0 (3 rondas perdidas), siendo el peor caso posible.
- El oponente se desconecta: Solamente se tendrán en cuenta los resultados de las rondas ya completadas, y las restantes se considerarán como victorias. Decidimos esta solución con tal de compensar al jugador por la desconexión del contrario, pero a la vez teniendo en cuenta su rendimiento con tal de no desequilibrar el sistema de forma excesiva.

3.5. Conclusión

A la hora de crear un sistema de matchmaking, el diseño de su funcionamiento es la parte más compleja. Esto se debe a la enorme cantidad de variables y flexibilidad a la hora de plantear su estructura, aunque se parta de una base ya estudiada. Las decisiones de diseño que se han tomado a lo largo de este capítulo definirán el mecanismo del sistema de cara a su implementación, y servirán para definir unos pasos a seguir durante el proceso de desarrollo.

Capítulo 4

Implementación

En el capítulo anterior se describió el diseño del sistema de *matchmaking*. Para implementarlo, entran en juego varios elementos (figura 4.1):

- Base de datos: La base de datos en la que se almacenará el historial de partidas de los jugadores (tanto las ya procesadas para la actualización de la puntuación, como las que aún están pendientes), su puntuación y RD, credenciales, y demás datos que se deseen guardar sobre un jugador.
- Servidor de actualización: Un servidor independiente que se encargará de actualizar las puntuaciones cada periodo de actualización (en nuestro caso, cada hora).
- Servidor de *matchmaking*: Un servidor con dos funciones: emparejar jugadores, y gestionar la conexión entre el cliente y la base de datos (enviar resultados de una partida, pedir información de un jugador, etcétera).
- Cliente: El cliente del juego, que se conectará al servidor y realizará las peticiones necesarias para jugar.

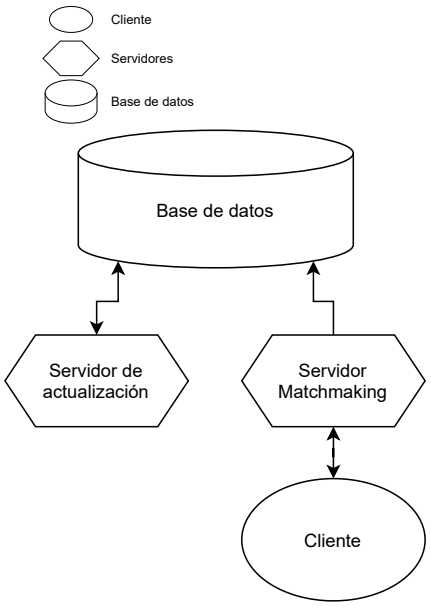

Figura 4.1: Esquema de servidores

A la hora de implementar este diseño, lo más importante es mantener cada componente independiente del resto, de manera que de haber algún fallo no colapse todo el sistema. Esto permite también modificarlo de forma más sencilla, de querer ajustarse distintos parámetros o cálculos.

El servidor de *matchmaking* se creará como un servicio REST, y este se conectará a la base de datos mediante un driver oficial.

4.1. Herramientas utilizadas

Para implementar todo, hemos hecho uso de las siguientes herramientas:

- MongoDB
  - Base de datos de tipo NoSQL que se utiliza para el almacenamiento de datos, relativos tanto a la gestión de cuentas como a los datos a emplear a la hora de realizar los emparejamientos.
- Node.js
  - Entorno de ejecución multiplataforma ideado para la creación de servicios web, base de todos los servidores REST.
- MongoDB Node.js Driver
  - Marco de aplicación para Node.JS, que permite acceder a la base de datos de MongoDB.
- Express.js
  - Framework para aplicaciones web de Node.JS, que permite crear la API REST de los servidores.

4.2. Servidor de matchmaking

El servidor está programado con Node.js, y la conexión se realiza mediante una API de tipo REST. REST, o *Representational State Transfer*, es un estilo de arquitectura software creado para estandarizar el desarrollo de la *World Wide Web*, aunque hoy en día se emplea más para referirse a APIs que obtienen información o ejecutan operaciones mediante HTTP. De las pautas que define, las más importantes son:

- Emplear un conjunto de operaciones buen definidas. En el caso de HTTP se definen varias, de las cuales cuatro son las más importantes:
  - POST, que sirve para enviar información.
  - GET, que sirve para pedir información.
  - PUT, que sirve para actualizar información.
  - DELETE, que sirve para eliminar información.

- Emplear una sintaxis universal para identificar los recursos. En el caso de un sistema REST se realiza mediante su URL, de forma que solo se pueda acceder al recurso mediante esta.

Su función principal es gestionar el proceso de emparejado de jugadores mediante una serie de servicios, a los cuales se acceden mediante *endpoints*, que en el caso de REST se tratan de URLs. Estos servicios reciben y envían información mediante documentos JSON.

Tenemos tres servicios principales: uno que gestiona las cuentas, otro que gestiona el *matchmaking* y por último, uno que nos permite realizar una gestión de versiones.

Un punto importante es que estos servicios son independientes del juego, a falta de la restricción de nuestro sistema de *matchmaking*, que los juegos contengan partidas de uno contra uno. Esto se debe a que estos servicios nunca requieren información específica de la implementación del juego, por lo que no es necesario modificar las funciones para adaptarlo. Incluso en el caso de enviar resultados al final de una partida, el servidor en ningún momento necesita leer la información contenida: se limita a enviarla a la base de datos, que la acepta (nuevamente) sin ninguna necesidad de acceder a sus contenidos.

Aun así, esto no significa que permita que se gestionen varios juegos a la vez. Es genérico en el sentido de que puede aplicarse a cualquier juego que se ajuste a la restricción de partidas uno contra uno, pero no en el sentido de que permita gestionar múltiples juegos distintos en una misma instancia.

En este caso, el sistema es reutilizable, con los parámetros de puntuación y desviación empleados por defecto (según la referencia de Glicko) que además pueden ser modificados por el desarrollador, al ser transparente en este sentido. De la misma manera, también puede modificarse el valor del periodo de actualización para adaptarlo a casos con partidas de diferente duración.

4.2.1. Definiciones

Antes de hablar de los servicios, conviene definir varios conceptos que se emplearán a lo largo de esta sección:

- Autenticación: Proceso por el cual se valida el usuario de una petición sin necesidad de enviar los credenciales en cada petición. Para ello se emplean *JSON Web Tokens*, cadenas encriptadas que contienen datos de un usuario (en nuestro caso, su número de identificación y su nombre de usuario) para diferenciarlas, que se generan por primera vez al hacer un inicio de sesión. Hay dos tipos de tokens:
  - Un *auth token*, que es válido durante un tiempo limitado (5 minutos en nuestro caso) y es el que se usará para validar las peticiones. Se ha fijado un límite de tiempo para evitar que puedan reutilizarse *tokens* de sesiones cerradas.
  - Un *refresh token*, que contiene la misma información que el anterior, pero no tiene límite de tiempo. Este *token* sirve para generar un *auth token* nuevo cuando el anterior haya caducado, pero no sirve para validar peticiones.
- Parámetros: Es la información enviada al hacer una petición. Esta información puede ofrecerse de tres formas distintas:

- *Body*: se envía un objeto JSON en el cuerpo de la petición, con los parámetros deseados. Se emplea solo en peticiones POST, y es el método por defecto.
- *Query*: se envía información al final de la URL. El formato es **"URL/?parametro1=valor1&parametro2=valor2"**, donde:
  - "URL/" es la URL del servicio.
  - "?" indica el comienzo del query.
  - "parametroN=valorN" indica el nombre y valor de un parámetro.
  - "&" separa distintos parámetros.
- *Path*: se envía información como parte de la URL. El formato es **"URL/:parametro/RESTO"**, donde:
  - "URL/" es la porción de la URL previa al parámetro.
  - ":parametro" indica que el valor indicado en esa sección de la URL será el valor del parámetro "parametro".
  - "RESTO" es la porción de la URL posterior al parámetro. Puede estar vacío, o pueden incluirse más parámetros de este estilo.

- **Respuestas**: Es la información recibida tras hacer una petición. Las respuestas siempre vienen indicadas con un código de error, y contienen la información en su cuerpo (o *body*). Los códigos distintos de 200 se consideran errores, y podemos definir dos casos que se dan siempre por defecto:
  - Código 200: por defecto no se recibirá nada en el cuerpo de la respuesta, considerando el código suficiente respuesta. Se especificarán los casos en los que sí se envíe algo.
  - Código distinto de 200: se recibe una respuesta del formato *{message: string}*, donde *message* es una cadena con una explicación del error.

  También existe una serie de códigos de error que se lanzan de forma general desde los servicios:
  - *401*: en un recurso que requiere autenticación, no se ha proporcionado un *auth token* y por tanto no ha podido autorizarse la petición.
  - *403*: en un recurso que requiere autenticación, se ha proporcionado un *auth token* inválido y por tanto no ha podido autorizarse la petición.
  - *502*: la base de datos no acepta conexión.

Definidos estos conceptos, damos paso a la documentación de los servicios REST.

4.2.2. */accounts*

Este servicio engloba todas las peticiones relacionadas con la gestión de cuentas (creación, borrado, petición de datos) y sesiones (inicio y cierre de sesión), además de encargarse de generar nuevos *auth tokens* y de subir los resultados de una partida al historial de un jugador. Mientras no se indique lo contrario estos servicios requieren autorización, y utilizarán la información del *token* para realizar sus tareas.

- **POST** */accounts*: crea una nueva cuenta de usuario. No requiere autenticación.
  - *Parámetros*:
    - *nick* (obligatorio | string): nombre de usuario.
    - *email* (obligatorio | string): e-mail del usuario.
    - *password* (obligatorio | string): contraseña del usuario. Esta contraseña se envía de forma segura al aplicar previamente el algoritmo SHA-256, de forma que en la base de datos nunca se guarde como texto plano.
  - *Respuestas*:
    - *200*: petición completada correctamente.
- **DELETE** */accounts*: permite eliminar la cuenta de usuario.
  - *Respuestas*:
    - *200*: petición completada correctamente.
- **GET** */accounts/check-availability*: permite verificar si un e-mail y/o un nombre de usuario proporcionados están en uso. No requiere autenticación.
  - *Parámetros* (debe proporcionarse al menos uno):
    - *nick* (obligatorio* | query | string): nombre de usuario.
    - *email* (obligatorio* | query | string): e-mail del usuario.
  - *Respuestas*:
    - *200*: petición completada correctamente.
      - *emailAvailable*: indica si está libre el e-mail proporcionado.
      - *nickAvailable*: indica si está libre el nombre de usuario proporcionado.
- **POST** */accounts/sessions*: verifica las credenciales proporcionados y, de ser válidas, abre una sesión (*login*) y genera un *auth token* y un *refresh token* para el usuario. No requiere autenticación.
  - *Parámetros*:
    - Debe proporcionarse al menos uno:
      - *nick* (obligatorio* | string): nombre de usuario.
      - *email* (obligatorio* | string): e-mail del usuario.
    - *password* (obligatorio | string): contraseña cifrada del usuario.
  - *Respuestas*:
    - *404*: no se ha encontrado un usuario con esos credenciales.
    - *200*: petición completada correctamente.
      - *id*: id del usuario.

        - ◇ *accessToken*: *access token* de esta sesión del usuario, que expira tras 5 minutos.
        - ◇ *refreshToken*: *refresh token* de esta sesión del usuario.

- **DELETE** */accounts/sessions*: permite cerrar una sesión (*logout*), e invalida tanto el *auth token* como el *refresh token* proporcionados.
    - *Parámetros*:
        - *refreshToken* (obligatorio | string): *refresh token* de la sesión a cerrar.
    - *Respuestas*:
        - *200*: petición completada correctamente.
- **POST** */accounts/sessions/refresh*: dado un *refresh token*, si este se corresponde a una sesión aún abierta, permite generar un *auth token* nuevo para renovar una sesión caducada. Realiza autenticación mediante el *refresh token*.
    - *Parámetros*:
        - *refreshToken* (obligatorio | string): *refresh token* de la sesión a renovar.
    - *Respuestas*:
        - *200*: petición completada correctamente.
            - ◇ *accessToken* (string): nuevo *access token* para esta sesión, que expirará otra vez tras 5 minutos.
- **GET** */accounts/by-id/{id}*: permite pedir información acerca de un jugador, dado su id. No requiere autenticación.
    - *Parámetros*:
        - *id* (obligatorio | path | int): id del usuario a buscar. Se incluye en la URL.
    - *Respuestas*:
        - *404*: no se ha encontrado un usuario.
        - *200*: petición completada correctamente. La respuesta consiste en toda la información del usuario almacenada en la base de datos, quitando el id interno de la base de datos e información importante como su contraseña cifrada (mediante la aplicación del algoritmo SHA-256, mencionado previamente) y su correo electrónico. El formato de esta información se detalla en la sección 4.4 *Base de datos: MongoDB*
- **GET** */accounts/by-nick/{nick}*: permite pedir información acerca de un jugador, dado su nombre de usuario. No requiere autenticación.
    - *Parámetros*:
        - *nick* (obligatorio | path | string): nombre del usuario a buscar. Se incluye en la URL.
    - *Respuestas*:

        - *404*: no se ha encontrado un usuario.
        - *200*: petición completada correctamente. La respuesta consiste en toda la información del usuario almacenada en la base de datos, quitando el id interno de la base de datos e información importante como su contraseña y su correo electrónico. El formato de esta información se detalla en la sección 4.4 *Base de datos: MongoDB*

- **POST** */accounts/rounds*: permite subir la información de una partida (resultados, otros datos que se hayan recopilado, etc.) a la base de datos, añadiéndose al historial de un usuario. El diseñador puede definir el formato de esta información como desee.
    - *Respuestas*:
        - *200*: petición completada correctamente.

4.2.3. */matchmaking*

Este servicio engloba todas las peticiones relacionadas con el proceso de emparejado. Todos estos servicios requieren autorización, y utilizarán la información del *token* para realizar sus tareas.

- **POST** */matchmaking*: añade al usuario a la lista de espera. Se realiza en su propio servicio por seguir el estándar de REST.
    - *Parámetros*:
        - *waitTime* (query | float): indica al servidor el tiempo (en milisegundos) que lleva esperando un jugador antes de entrar en la lista de espera. Tiene 0 como valor por defecto.
    - *Respuestas*:
        - *200*: petición completada correctamente.
- **GET** */matchmaking*: toma el usuario y, de estar en la lista de espera, trata de emparejarlo con otro, realizando además una limpieza de la lista (eliminando usuarios que lleven mucho tiempo sin realizar una petición).
    - *Parámetros*:
        - *waitTime* (query | float): indica al servidor el tiempo (en milisegundos) que lleva esperando un jugador antes de entrar en la lista de espera. Tiene 0 como valor por defecto.
    - *Respuestas*:
        - *404*: el usuario indicado no se halla en la lista de espera.
        - *200*: petición completada correctamente.
            - *found* (bool): indica si se ha encontrado un rival óptimo. En caso de ser falso, los parámetros restantes no se definen.
            - *finished* (bool): en caso de que este usuario aún no haya sido emparejado con otro usuario, devuelve falso. En caso contrario, verdadero.

- ◇ *rivalID* (int): id del rival encontrado.
- ◇ *rivalNick* (string): nombre de usuario del rival encontrado.
- ◇ *bestRivalRating* (float): puntuación del rival encontrado.
- ◇ *bestRivalRD* (float): RD del rival encontrado.
- ◇ *myRating* (float): puntuación del usuario en ese momento.
- ◇ *myRD* (float): RD del usuario en ese momento.

- **DELETE** */matchmaking*: elimina el usuario de la lista de espera. Se llama en caso de cancelar la búsqueda de pareja o de haberse encontrado.
  - *Respuestas*:
    - *404*: el usuario indicado no se halla en la lista de espera.
    - *200*: petición completada correctamente.

4.2.4. */version*

Este servicio solo cuenta con una petición, cuyo objetivo es ofrecer un control de versiones. No requiere autorización.

- **GET** */version*: proporciona el número de versión del juego designada en el servidor. Sirve para evitar que jugadores con versiones antiguas se conecten con otros jugadores, para evitar errores.
  - *Respuestas*:
    - *200*: petición completada correctamente.
      - ◇ *version* (string): la versión actual guardada en el servidor.

4.3. Servidor de actualización

Las actualizaciones de puntuaciones se realizan en el servidor de actualización el cual, cada hora, realiza una petición a la base de datos de los jugadores con rondas pendientes de cálculo y las procesa, realizando las operaciones necesarias según los datos del oponente. Estas puntuaciones son guardadas en la base de datos, en el documento de cada jugador.

En esencia, este servidor solamente requiere de los usuarios sus partidas (con el tiempo y resultado de cada ronda, además del identificador del rival) y sus clasificaciones. Así pues, mientras se ofrezca esta información, el contenido restante del documento de los jugadores dará igual.

Un problema importante a la hora de actualizar puntuaciones es qué sucede si se suben partidas al historial durante el proceso de actualización. Para evitar eliminar o saltarnos partidas del historial, hemos decidido que solamente se cogerán aquellos jugadores que tengan partidas pendientes al empezar la actualización, archivando estas partidas pendientes y usando una copia para realizar los cálculos. Si bien esto significa que puede haber

jugadores que no se actualicen, o que solo uno de los oponentes en una partida se actualice en un periodo, consideramos que es una pérdida aceptable frente a, potencialmente, perder información o sobrescribirla. Además, dados los cortos periodos de actualización esta diferencia se arreglaría rápidamente.

4.4. Base de datos: MongoDB

Para almacenar los datos necesarios de cada usuario hemos decidido emplear una base de datos NoSQL, en concreto MongoDB. La decisión de emplear esta en concreto se debe a varios factores:

- Acepta de forma nativa el formato JSON para cualquier tipo de acción, simplificando la comunicación entre la misma y la API REST que empleamos como servidor.
- Es una base de datos no relacional, o NoSQL. Esto significa que permite manejar cantidades de datos enormes (de cara a guardar los datos de todos los jugadores, era un factor importante) sin necesidad de establecer una estructura fija, y que es sencillo replicar una misma base de datos en varias máquinas de bajo coste, en caso de necesitarse ampliar el sistema o realizar una copia de seguridad.
- Es fácil de implementar e integrar en distintos lenguajes, teniendo *drivers* oficiales que realizan gran parte del procesado trasero necesario para enviar información.

Esta base de datos podrá residir en la nube o en la misma máquina en la que se ejecuten los servidores, y contará con dos colecciones:

- *data*: Esta colección guarda en un único documento información pertinente al funcionamiento de los diversos sistemas de gestión de datos:
  - *playerCount*, un contador con la cantidad total de usuarios registrados, necesario a la hora de asignar los identificadores de usuario (es más rápido y fiable acceder a un documento que guarde esta información, frente a hacer un recuento de todos los jugadores cada vez que se necesite este valor).
  - *dateLog*, un historial de fechas en las que se actualizaron las puntuaciones.
  - *lastT*, un contador que aumenta en 1 con cada periodo de actualización realizado, y se emplea para calcular la desviación de los jugadores. Es igual que la cantidad de entradas en dateLog.
- *tefege*: Esta es la colección que contiene los documentos con la información de cada jugador, ordenados por ID mediante un índice. Estos documentos pueden contener toda la información que desee el desarrollador, pero los parámetros mínimos requeridos son:
  - **id**, el identificador numérico de este usuario.
  - **nick** y **email**, el nombre de usuario y su correo electrónico.
  - **password**, la contraseña cifrada mediante la aplicación del algoritmo SHA-256, descrito previamente. Este cifrado se realiza en una sola dirección, de forma que la base de datos nunca debe saber la contraseña real.

- **creation**, **lastLogin** y **lastGame**, cadenas que indican las fechas de creación de cuenta, de último inicio de sesión y de la última partida jugada respectivamente.
- **rating** y **RD**, la puntuación y desviación (respectivamente) empleadas en Glicko.
- **lastT**, un número que indica el último periodo de actualización en el que se procesó el historial de este jugador.
- **pending**, un historial de partidas pendientes de ser procesadas. Solo se actualizarán jugadores que tengan información en este historial.
- **history**, un historial de partidas ya procesadas. En cada periodo de actualización se volcarán los contenidos de **pending** aquí. Útil en caso de querer actualizar o sustituir el sistema de clasificación, o en caso de querer ajustarse el cálculo de distintos parámetros.

Cabe destacar que en la implementación del servidor de *matchmaking* se ofrece la posibilidad de definir una función de procesado de resultados. Por defecto, cuando el servidor recibe una petición de envío de los resultados de una partida, el *driver* de MongoDB se limita a subirlo al historial del jugador correspondiente. Pero en caso de desearse realizar algún tipo de procesamiento previo (recontar victorias, derrotas y empates, procesar información recogida durante una partida para su posterior análisis, etcétera), el cual debe ser realizado por MongoDB, se permite redefinir esta función para el procesado y posterior envío de datos.

4.5. Testeo del sistema de *matchmaking*

Una vez implementado el sistema, se decidió probarlo con una simulación para comprobar que todo funcionara según lo esperado, de forma previa a integrarlo con el caso de estudio.

4.5.1. Objetivo

Nuestro objetivo es comprobar que el sistema funciona correctamente, tanto por la parte de Glicko como por la de de los servidores de *matchmaking* y de actualización, buscando fallos o agujeros en la implementación y arreglándolos.

4.5.2. Metodología

Las pruebas se llevarán a cabo a través de una simulación de jugadores en busca de partidas, completándolas y enviando los resultados para después buscar una nueva partida o desconectarse y ser sustituido por otro jugador.

Así pues, el proceso de simulación consiste de:

- **Generar jugadores**: Hace falta generar una serie de jugadores ficticios e introducirlos en la base de datos, con puntuaciones y desviaciones distintas de las ofrecidas por defecto para simular una base de jugadores variada.

- **Añadir jugadores a la lista de espera**: En vez de sumar todos los jugadores generados, se especificará una cantidad aleatoria que insertar. Después se añadirán estos jugadores a la lista de espera si no están ya presentes.
- **Realizar peticiones de emparejado**: Se realizarán las peticiones de emparejado en bucle, incrementando un tiempo de espera interno mientras no se encuentre pareja.
- **Simular una partida**: De encontrarse una pareja, se eliminará a ambos jugadores de la lista de espera y se realizará una predicción del resultado de una partida según sus puntuaciones y desviaciones. Estos resultados se enviarán a la base de datos y se elegirá de forma aleatoria si los jugadores volverán a la lista de espera (simulando que quieren jugar otra vez) o si serán sustituidos (simulando que se desconectan y otro jugador diferente se conecta).

4.5.3. Proceso

Generar jugadores

Glicko define los valores "medios" en su sistema como:

- **Puntuación**: 1500.
- **Desviación de puntuación**: 350.

La puntuación "media" es la misma que la puntuación asignada a jugadores nuevos por defecto, mientras que una desviación de 50 representa un punto medio en las desviaciones, jugadores que no son ni muy esporádicos ni muy frecuentes. Nos interesa usar este valor medio para simular de una forma más fiable que se tratan de jugadores reales. Considerando estos valores, un nuevo jugador tendría un rango de habilidad de $1500 \pm 50 \times 2$.

Así pues, se generarán un total de 500 jugadores con puntuaciones y desviaciones siguiendo una distribución normal basada en estos dos valores "medios":

- **Puntuación**: *N(1500,200)* (fig. 4.2)
- **Desviación de puntuación**: *N(50,10)* (fig. 4.3)

Una vez se han generado estas puntuaciones y desviaciones, se asignan a un jugador con nombre, correo electrónico y contraseña aleatorios y se registran en la base de datos. (fig. 4.4)

Introducir jugadores a la lista de espera y realizar peticiones de emparejado

Tras un registro inicial de los jugadores, se inicia el servidor de *matchmaking* y se maneja la conexión de todos los jugadores desde un cliente ficticio conectado al servidor. El proceso que empleamos para testear selecciona una cantidad del total de jugadores existentes y comienza un bucle. En este bucle se añadirán aquellos jugadores que no estén en la lista de espera a la misma, y después se procederá a realizar peticiones de emparejado. Este bucle seguirá de forma indefinida, hasta que decidamos parar el testeo, y se incrementará un contador interno de "tiempo de espera" en los jugadores mientras no encuentren pareja.

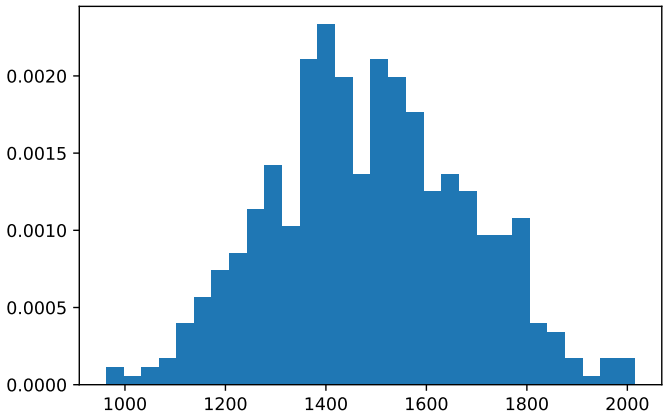

Figura 4.2: Gráfica que representa las puntuaciones generadas mediante distribución normal. El eje X representa rangos de puntuaciones, el eje Y representa la proporción de elementos en ese rango con respecto al total.

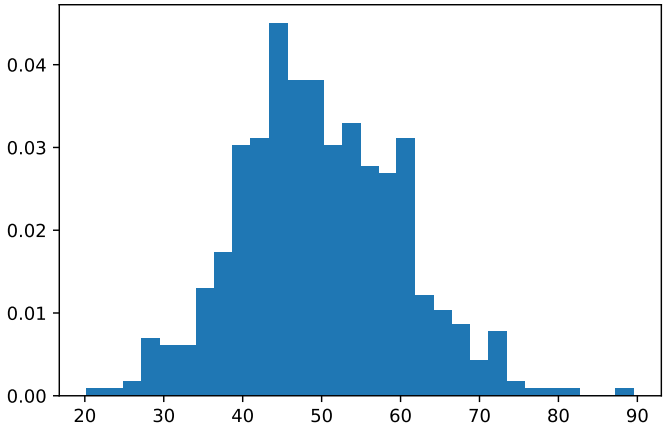

Figura 4.3: Gráfica que representa las desviaciones generadas mediante distribución normal. El eje X representa rangos de desviaciones, el eje Y representa la proporción de elementos en ese rango con respecto al total.


```
_id: ObjectId("60a15fa363a5cc131c1984a2")
nick: "Axe Got"
email: "agotc6@timesonline.co.uk"
password: "14qbXAt0zMq08"
salt: ""
creation: "Sun May 16 2021 20:08:35 GMT+0200 (hora de verano de Europa central)"
rating: 1482.4989511082613
RD: 42.16769065153492
id: 16
```


Figura 4.4: Ejemplo del documento de un jugador simulado en la base de datos.

Simulación de una partida

El bucle de peticiones seguirá ejecutándose hasta que dos jugadores se emparejen mutuamente, y entonces procederá a eliminarlos de la lista de espera y simular una partida entre ambos.

El proceso de simulación consta de los siguientes apartados:

- **Asignación de resultados**: Se calculan los resultados para las tres rondas que componen un partida. Para llevar a cabo este cálculo, primero se estima la probabilidad de victoria de ambos jugadores a través de la fórmula de predicción de Glicko (definida ya en el estado del arte, como se indica en la ecuación 2.7).

  Este valor representa la probabilidad de victoria del jugador seleccionado frente a su oponente, un valor decimal comprendido entre 0 y 1. Dada esta estimación, se genera otro valor entre 0 y 1. Si este nuevo valor es inferior al anterior se considera victoria para el jugador seleccionado, y si es mayor se considera derrota.

  Este proceso se realiza 3 veces, uno por cada ronda de la partida.

- **Enviar resultados al servidor**: Se envían los resultados de cada uno de los jugadores participantes al servidor, almacenando en cada uno de los datos de los jugadores un objeto que representa la partida y que contiene los resultados y tiempo de las tres rondas, además del id del oponente.

- **Desconexión aleatoria de jugadores**: Para cada uno de los jugadores en línea se almacena el total de partidas seguidas que lleva. Esto se utiliza para simular la desconexión del jugador tras llevar un tiempo jugando, calculando la probabilidad de desconexión P. (ecuación 4.1)

$$P = Math.Floor(Math.random(0,1) * (totalPartidas * 10)) \tag{4.1}$$

  En el caso de que se dé una desconexión, se introduce un nuevo jugador a la lista para mantener el mismo número de jugadores activos durante el proceso de pruebas.

4.5.4. Resultados

Una vez implementado el sistema de prueba y solventados los primeros fallos en términos de conexión, se pasó a estudiar los emparejamientos, donde se detectaron algunos errores críticos en el cálculo de los rangos de los jugadores. Nuevamente se fueron arreglando estos errores, hasta que los emparejamientos se realizaran de forma satisfactoria. Y una vez se comprobó que las parejas se realizaban correctamente, se procedió a probar el servidor de actualización, arreglando los errores que surgieran hasta que el código se ejecutara sin ningún problema.

Se dejó la simulación sola durante 10 minutos, generando aproximadamente 1 millón de partidas entre los jugadores, y después se forzó una actualización para asegurarnos que el sistema no tenía problemas de carga. (fig. 4.5)

El resultado, pues, fue verificar que el sistema funcionaba correctamente. Nos dimos cuenta de que hacía falta reestructurar los servicios de emparejado (4.2.3 */matchmaking*),

Figura 4.5: Ejemplo de la actualización de un jugador. Tras tres derrotas, se puede observar la puntuación disminuyendo, y al ser su primera actualización también disminuye su desviación.

además de cambiar el formato de guardado de partidas. Todos estos cambios ya están reflejados en la memoria.

4.6. Conclusión

Con esto queda completada la implementación del sistema de *matchmaking*, con todas sus partes. También se ha probado su correcto funcionamiento mediante una simulación, permitiendo encontrar y solventar errores de funcionamiento. Así pues, se puede proceder al siguiente paso, crear un caso de estudio mediante el cual se pueda probar esta implementación en un entorno con jugadores reales.

Capítulo 5

Caso de estudio

El sistema de matchmaking requiere de un videojuego sobre el que aplicarse, por eso mismo, este proyecto incluye el desarrollo de un pequeño videojuego multijugador sobre el que se pone a prueba el emparejamiento y la actualización de puntos de los jugadores.

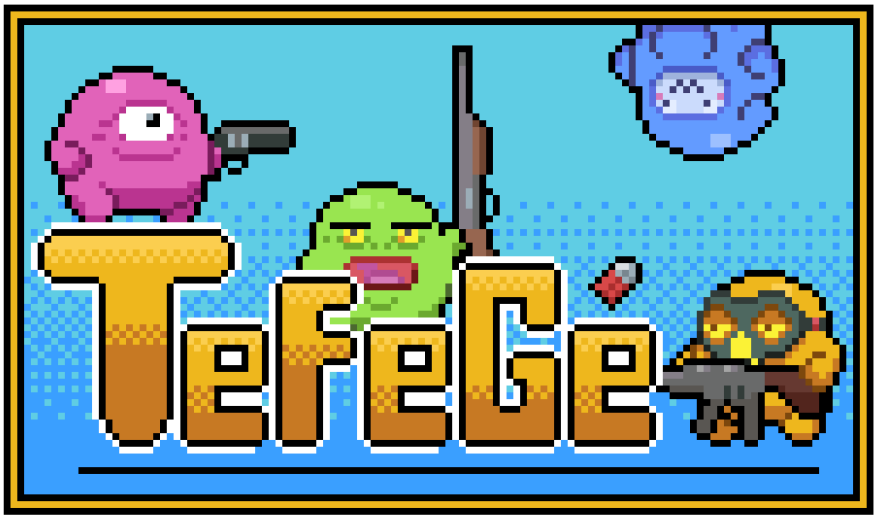

Figura 5.1: Logo del juego TeFeGe

TeFeGe se trata de un juego de género *twin-stick shooter* multijugador con perspectiva en vista cenital.

- *Twin Stick*: El juego se controla con un joystick para moverse y otro para apuntar (en caso de usar mando), o las teclas W/A/S/D para moverse y el ratón para apuntar (en caso de emplear teclado y ratón)
- *Shooter*: La jugabilidad se centra en el uso de armas que disparan a la hora de atacar al oponente.

Cada partida constará de 3 rondas diferentes, en las que 2 jugadores se enfrentarán en

un mapa de reducido tamaño (para forzar el enfrentamiento directo entre ambos) entre sí a lo largo de los 45 segundos que dura cada ronda, con el objetivo de eliminar al rival agotando sus puntos de vida, utilizando el arma y habilidad del personaje utilizado. En caso de alcanzar el límite de tiempo, se tratará como un empate.

Este aspecto se relaciona directamente con el propio sistema de *matchmaking*, tanto en la cuantificación del resultado de la partida como en el tiempo de actualización que fijaremos.

Considerando que las rondas duran 45 segundos como máximo, y cada partida consiste de 3 rondas, las 10 partidas recomendadas por periodo de actualización en un juego como el nuestro equivaldrían (aproximadamente) a 30 minutos. Estos 30 minutos son, por supuesto, asumiendo una cantidad puramente matemática, por lo que optamos por ampliarlo a 1 hora de espera entre actualizaciones.

De manera previa a comenzar la partida se pasará por un menú de selección de personajes en el que cada jugador elegirá su avatar para la partida. Con el fin de diversificar el juego y ofrecer experiencias diferentes, cada personaje cuenta con un arma y habilidad diferente, que pueden dar lugar a distintas estrategias y estilos de juego.

El resultado de estas rondas se empleará para actualizar el historial de partidas de cada jugador, además de obtener datos para el propio sistema de *matchmaking* (el cual es objetivo principal del trabajo). Es por esto que, en vez de establecer la condición de victoria de la partida al mejor de 3 (o lo que es lo mismo, con 2 victorias se podría dar un ganador), se obliga a completar las 3 rondas con el fin de poder obtener más datos de la misma, y poder lograr un sistema de *matchmaking* más completo y ajustado de cara a futuros emparejamiento, el cual se realiza en base a la puntuación de cada usuario antes de cada partida.

Desde el punto de vista del usuario, podría describirse este proceso en los siguientes pasos:

1 Al ejecutar el juego, se presenta el menú principal donde el usuario puede escoger entre múltiples opciones, entre ellas “Jugar”, que permite acceder al *lobby* multijugador.

2 Dentro del *lobby*, el usuario escogerá el personaje a emplear en las partidas. Una vez seleccionado, esperará a recibir un oponente del servidor de *matchmaking*.

3 Tras haber encontrado un oponente, se inicia la partida. Esta se compone de 3 rondas de 45 segundos como máximo. El jugador debe tratar de eliminar al oponente haciendo uso de las habilidades y armas del personaje escogido.

   Cuando uno de los jugadores ve su vida reducida a 0 o el tiempo termina, finaliza la ronda. La ronda será considerada ganada por aquel jugador que haya conseguido derrotar al contrincante o, en caso de terminarse el tiempo, resultará en un empate.

   Se realizarán las tres rondas independientemente del ganador total, con tal de recopilar datos para el *matchmaking*. Así, en caso de perder solamente 2 de las 3 rondas, se tendrá en cuenta.

4 Tras finalizar la partida, el jugador pasará a la pantalla de resultados donde podrá ver los detalles de la misma. Desde aquí, el usuario podrá regresar al menú principal

o podrá decidir jugar de nuevo una partida con el mismo personaje o regresar a la pantalla de selección de personaje para cambiarlo.

5.1. Estudio de mercado

Para conocer el público objetivo al que orientar el videojuego de cara a buscar usuarios para las pruebas, se han elaborado una serie de encuestas que detallen las preferencias de los potenciales jugadores.

El estudio se ha realizado tanto en español como en inglés, obteniendo un total de 207 y 9 respuestas respectivamente. Estas encuestas se difundieron por redes sociales y grupos de gente aficionada a los videojuegos. Dada la poca cantidad de respuestas en inglés, se ha decidido orientar el desarrollo principalmente para el interés de los usuarios españoles.

La encuesta debía probar nuestro alcance, así como recabar información acerca del tipo de público y su interés o afición por los videojuegos (fig. 5.2), detalles necesarios para utilizar como referencia a la hora de establecer el número y duración de las rondas.

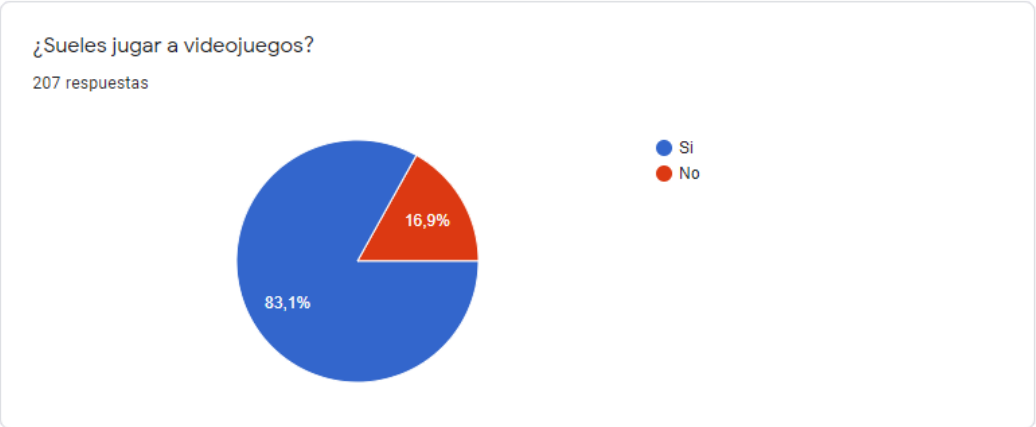

Figura 5.2: El objetivo de esta pregunta era determinar el interés del público alcanzado.

También era necesario obtener información acerca de la plataforma de desarrollo sobre la que se iba a plantear el proyecto. La información obtenida de la encuesta muestra una clara orientación hacia el uso de PC como plataforma distribuyéndolo mediante un instalador. (fig 5.3)

De manera complementaria, se decidió conocer qué repercusión tendría una posible elección de otras plataformas de desarrollo como teléfonos móviles (fig. 5.4) o navegadores web (fig. 5.5), para tener referencias acerca del posible impacto de un cambio de plataforma durante el desarrollo o el futuro del videojuego.

Por último, y anticipando que PC seguramente sería la plataforma elegida, se necesitaba información acerca de los controles preferidos por los usuarios por cuestiones de diseño como de equilibrado (fig. 5.6). Dados los resultados, se ha considerado que ambas opciones deberían estar disponibles presentes en el videojuego.

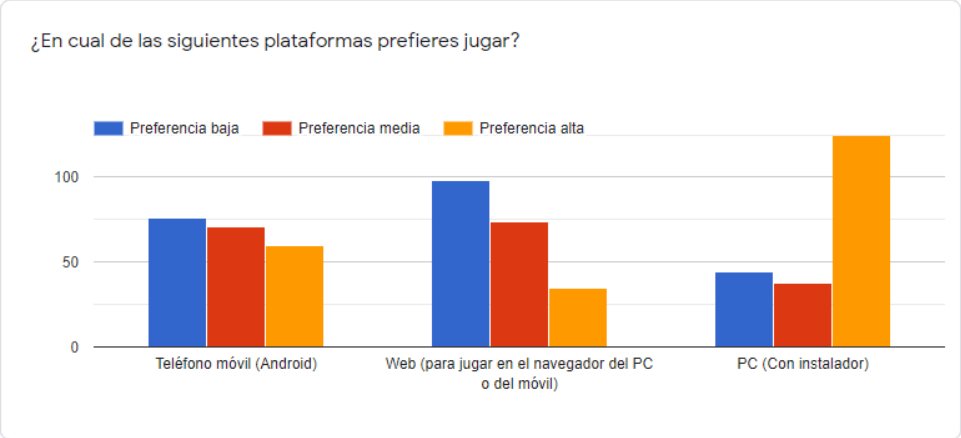

Figura 5.3: En esta pregunta pedimos a los encuestados que ordenaran tres plataformas según su preferencia: Android, Web y PC.

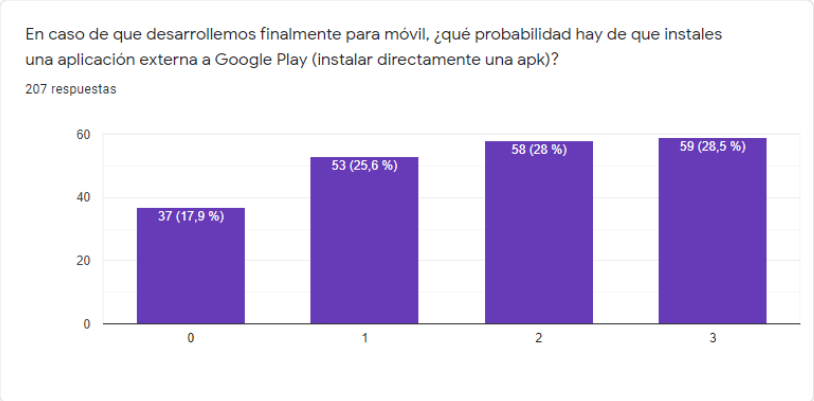

Figura 5.4: Frente a la posibilidad de plantear un desarrollo para Android, era necesario conocer cómo de importante sería subir el juego a la Google Play Store.

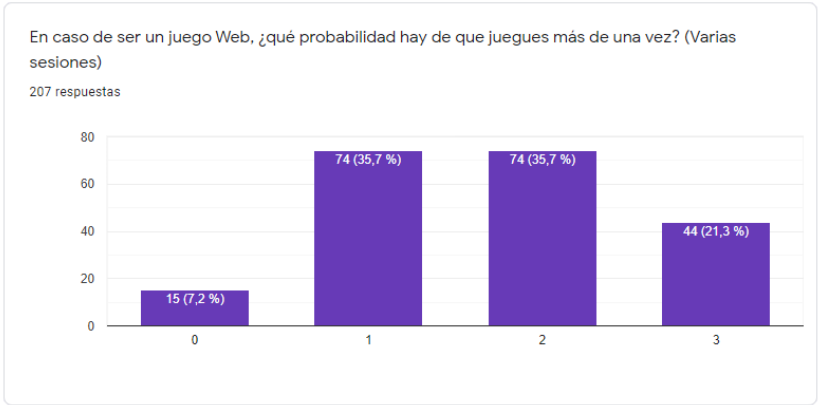

Figura 5.5: En caso de plantear un desarrollo para Web, es necesario conocer la frecuencia con la que jugarán los usuarios.

Figura 5.6: Esta pregunta informa sobre el método de control preferido por el público.

5.2. Inspiraciones

A la hora de

- Nuclear Throne[1]: Desarrollado por Vlambeer, además de la parte estética en referencia al mapa, diseño de personajes y armas,también ha sido usado como referente en su jugabilidad frenética, que hemos intentado imitar en un título multijugador.

Figura 5.7: Nuclear Throne

- Enter The Gungeon[2]: Desarrollado por Dodge Roll y publicado en el año 2016, se trata de un caso similar al anterior juego en términos de aspecto visual y jugable, los cuales también hemos utilizado como fuente de inspiración, especialmente en este caso la relación visual entre personaje y arma.

[1]https://store.steampowered.com/app/242680/Nuclear_Throne/
[2]https://store.steampowered.com/app/311690/Enter_the_Gungeon/

Figura 5.8: Enter The Gungeon

- Brawl Stars[3]: Título desarrollado para móviles por parte de Supercell, empleado como referencia de cara al diseño de la interfaz con el objetivo de ser poco intrusiva para el jugador y a la par, sencilla de comprender a primera vista. También se ha observado y planteado esquemas de controles inspirados en este juego de cara a una posible versión para teléfonos móviles que finalmente fue descartada.

Figura 5.9: Brawl Stars

5.3. Mecánicas

Las mecánicas principales del juego son:

- **Disparos y habilidades**: Cada personaje tiene un arma con sus propias balas, cadencia de disparo, munición, etcétera. Similarmente, todos cuentan con una habilidad especial propia que requiere esperar un tiempo de recarga con cada uso. La recarga de las armas es automática, realizándose inmediatamente una vez se queden sin munición. No se puede recargar el arma manualmente.
- **Movimiento y apuntado de arma**: El movimiento del personaje y el apuntado del arma son independientes el uno del otro, una característica propia del género.

[3]https://play.google.com/store/apps/details?id=com.supercell.brawlstars&hl=es&gl=US

- **Rondas de tiempo limitado**: Las partidas se componen de 3 rondas de 45 segundos, y cuando se acaba el tiempo se considera un empate y empieza la siguiente ronda.

5.4. Interfaz de usuario

El diseño de interfaz busca ser lo más sencillo posible para evitar posibles distracciones en medio de la acción. Por ello, únicamente se muestra la información necesaria sobre el personaje controlado, siempre situado en el centro de la cámara. La interfaz se divide en dos secciones: la interfaz del personaje, y el contador de tiempo.

La interfaz del personaje (fig. 5.10) se aplica tanto al controlado por el jugador como al del oponente, y se compone de:

- Barra de puntos de vida (subdividida en barras verdes que indican cada punto de vida), de color amarillo y colocada sobre la barra de munición.
- Barra de munición (subdividida en barras que indican cada bala restante), de color gris y colocada bajo la barra de vida, ajustado en proporción a la capacidad de munición del arma del personaje correspondiente.
- Estado de la habilidad especial, de forma circular y a la izquierda de las otras dos, en color verde cuando puede ser empleada, en negro cuando se ha utilizado, y con una animación que indica el progreso de la recarga de la misma.
- Nombre de usuario.

Figura 5.10: Ejemplo de la interfaz del personaje.

Por su parte, el contador de tiempo (fig. 5.11) se compone de:

- Contador de tiempo, que indica los segundos restantes en la ronda.
- Indicadores de resultados, tres cápsulas grises que se colorean de rojo, amarillo y verde según si la ronda que representan ha sigo perdida, empatada o ganada,

respectivamente. Las cápsulas correspondientes a rondas no completadas se dejan en gris.

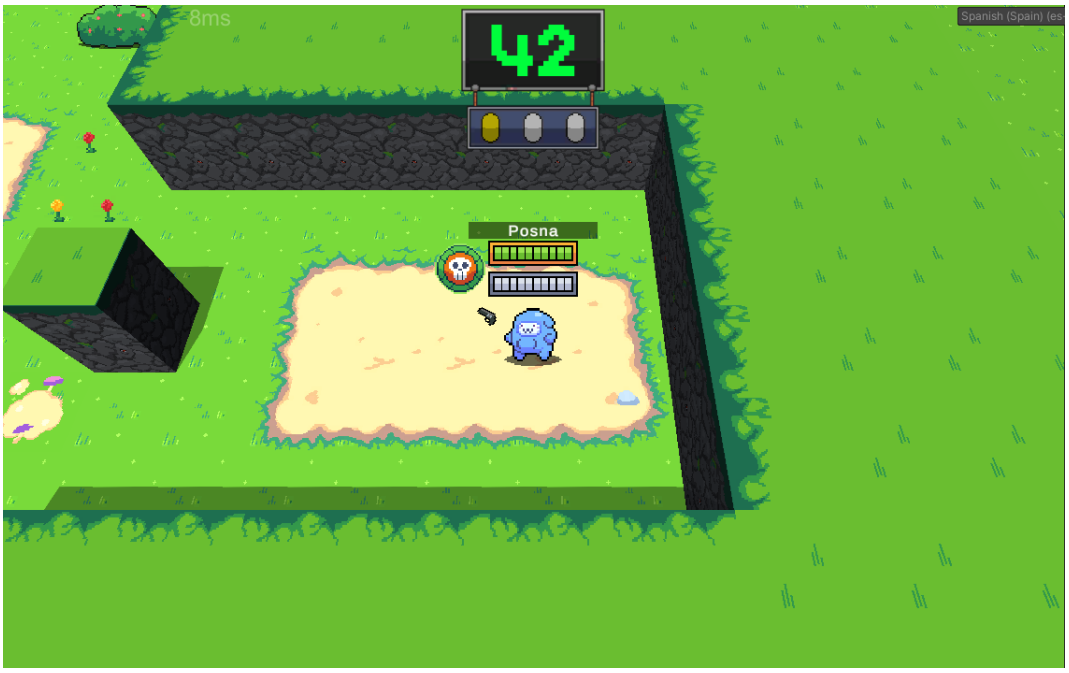

Figura 5.11: Ejemplo de la interfaz completa del juego.

De cara a comprobar que la interfaz cumplía con nuestros objetivos, hicimos una serie de pruebas de usuario con personas externas al desarrollo, con el objetivo de determinar la claridad de la misma y su efectividad a la hora de transmitir la información.

Para comprobar la funcionalidad y claridad de la interfaz, se realizaron una serie de pruebas de usuario para probar su efectividad. Los usuarios debían observar capturas de la interfaz para identificar los elementos. Posteriormente se les puso a jugar unos minutos y se repitió la pregunta, viendo las diferencias entre las respuestas. También se les realizó una entrevista para determinar la comprensión de las mecánicas, de la interfaz, reacciones al diseño visual, etcétera.

A lo largo de estas pruebas, además de mencionarse detalles de corte más estético como la coherencia entre estilo de menús e interfaz en las partidas, se destacan aspectos como que el tamaño es correcto (permite observar con claridad los distintos elementos sin que lleguen a molestar), además de que su identificación, incluso sin referencias del comportamiento en partida, resulta generalmente positiva.

5.5. Jugabilidad

5.5.1. Control del juego

El control del juego se realizará mediante teclado y ratón, o mando, siguiendo el esquema detallado en la figura 5.12.

- Para apuntar el arma del personaje elegido, se hará uso del ratón o el joystick derecho en caso de utilizar mando.

Figura 5.12: Esquema de controles del juego.

- Para moverse por el mapa, se hará uso de las teclas W (arriba), A (izquierda), S (abajo) y D (derecha) en caso de utilizar teclado, o del joystick izquierdo, si se opta por el mando.
- Para disparar, se emplearán el click izquierdo del ratón, o los botones RB y RT del mando (indistintamente).
- Para usar la habilidad del personaje, se emplearán el click derecho del ratón, o los botones LB y LT.

5.5.2. Personajes

El juego consta de cinco personajes, cada uno con un arma y una habilidad especial únicas para permitir diferentes estilos de juego. Todos los personajes cuentan con 10 puntos de vida y (salvo un caso) todas las balas funcionan de forma similar, desapareciendo una vez colisionen o se pase su tiempo de vida (para diferenciar armas de largo y corto alcance).

Figura 5.13: *Sprites* de Manolo McFly

- **Manolo McFly**:
  - Arma: Escopeta
    - Disparo semi-automático, con corto alcance
    - Daño: 1 punto de vida por bala
    - Munición: 9 balas por cargador

- Dispara 3 proyectiles en cono, haciendo que pueda hacer mucho daño en poco tiempo, pero con recargas frecuentes

- Habilidad: Cóctel molotov
  - Lanza un cóctel molotov que deja un charco de fuego donde impacte
  - En caso de golpear a un jugador se rompe, pero no hace daño
  - El charco desaparece pasados unos segundos, y hace daño a los jugadores que entren en contacto con el mismo

Figura 5.14: *Sprites* de Chuerk Chuerk

- **Chuerk Chuerk**:
  - Arma: Ametralladora
    - Disparo automático, con alta cadencia, poca precisión (ya que las balas se dispersan de su trayectoria con cierta aleatoriedad), pero a cambio cuenta con un alcance elevado.
    - Daño: 1 punto de vida por bala.
    - Munición: 10 balas por cargador
  - Habilidad: Turbocuesco
    - Al contrario que las otras habilidades, tiene su propia carga
    - Mantener pulsado el botón de habilidad dará un impulso de velocidad hasta que bien se suelte el botón, o bien se acabe la carga de la habilidad
    - Activar la habilidad deja un rastro de nubes verdes a su paso, que causan daño a los enemigos, pero desaparecen pasados unos segundos

Figura 5.15: *Sprites* de Bob Ojocojo

- **Bob Ojocojo**:
  - Arma: Pistola de goma
    - Disparo automático, con cadencia media y precisión media (dispersión reducida), con alto alcance.
    - Daño: 1 punto de vida por bala.
    - Munición: 5 balas por cargador.

- Las balas rebotan contra muros y obstáculos hasta 3 veces, desaparecen al siguiente, o bien tras acabarse su tiempo de vida.

- Habilidad: *Sniper*
  - Mientras se tenga pulsado el botón de habilidad, la cámara se aleja para mostrar el mapa entero y se muestra un gráfico que indica la dirección de disparo.
  - Al soltar el botón, se dispara una bala de alta velocidad y daño elevado que no desaparece hasta colisionar con algún elemento del mapa.

Figura 5.16: *Sprites* de Camomila Séstima

- **Camomila Séstima**:
  - Arma: Revólver
    - Disparo semi-automático, con alto alcance.
    - Daño: 1 punto de vida por bala.
    - Munición: 6 balas por cargador.
  - Habilidad: *Dodge*
    - Activar la habilidad hace que Camomila se deslice en la dirección en la cual se estaba moviendo previamente durante un breve periodo de tiempo.
    - Mientras se desliza, no puede recibir daño.
    - Al detenerse, se dispara toda la munición restante en la dirección a la cual esté apuntando, en forma de una nube de balas.

Figura 5.17: *Sprites* de Bad Baby

- **Bad Baby**:
  - Arma: Micrófono
    - Disparo semi-automático, con alto alcance y una velocidad menor
    - Daño: 1 punto de vida
    - Munición: 5 balas por cargador
    - Las balas flotan en círculo alrededor de Bad Baby

- Estas balas hacen tanto de escudo (protegiendo de balas que contacten con estas) como de munición, en ambos casos eliminando una bala del escudo

- Las balas situadas en el escudo no dañan a otros jugadores

- Habilidad: Seducción

  - Igual que la habilidad *Sniper*, mantener pulsado el botón de habilidad abre un indicador de la dirección en la que se disparará, si bien no se hace zoom.

  - Soltar la habilidad disparará un corazón (a una velocidad reducida) que, al chocar con un jugador, lo fuerza a moverse hacia Bad Baby durante unos segundos.

5.5.3. Niveles

El juego cuenta con un único mapa de tamaño reducido con distintas coberturas que da lugar tanto a que los jugadores puedan enfrentarse de manera directa como persiguiéndose para sorprender al contrario.

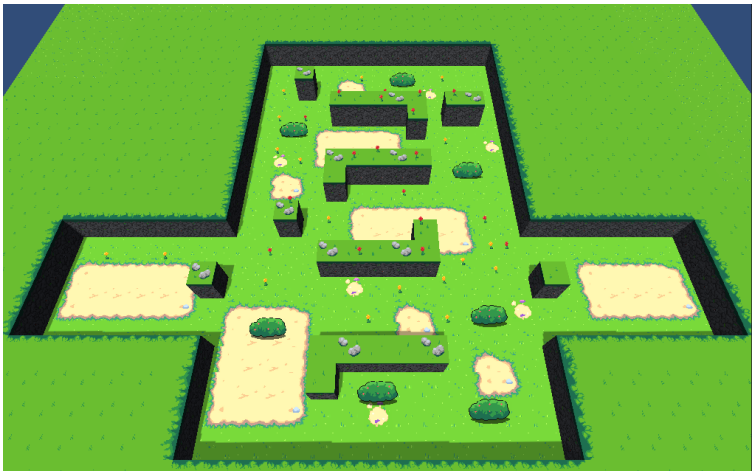

Figura 5.18: Mapa del juego

5.6. Menús y flujo de juego

Buscando ofrecer una experiencia de usuario sencilla, y dada la simplicidad del juego, los menús son poco complejos y muestran la información de forma concisa. (fig. 5.19).

El flujo de juego completo es el detallado en la figura 5.20.

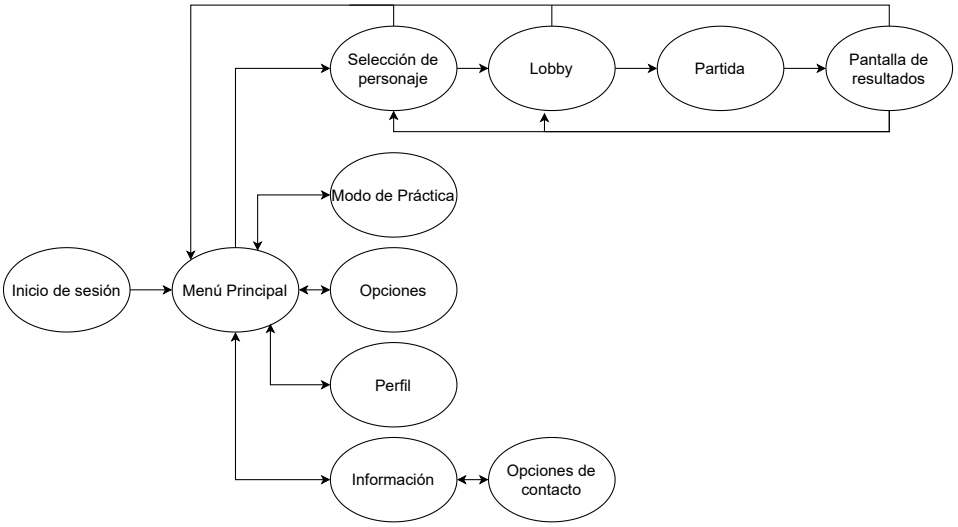

Figura 5.19: Menús del juego

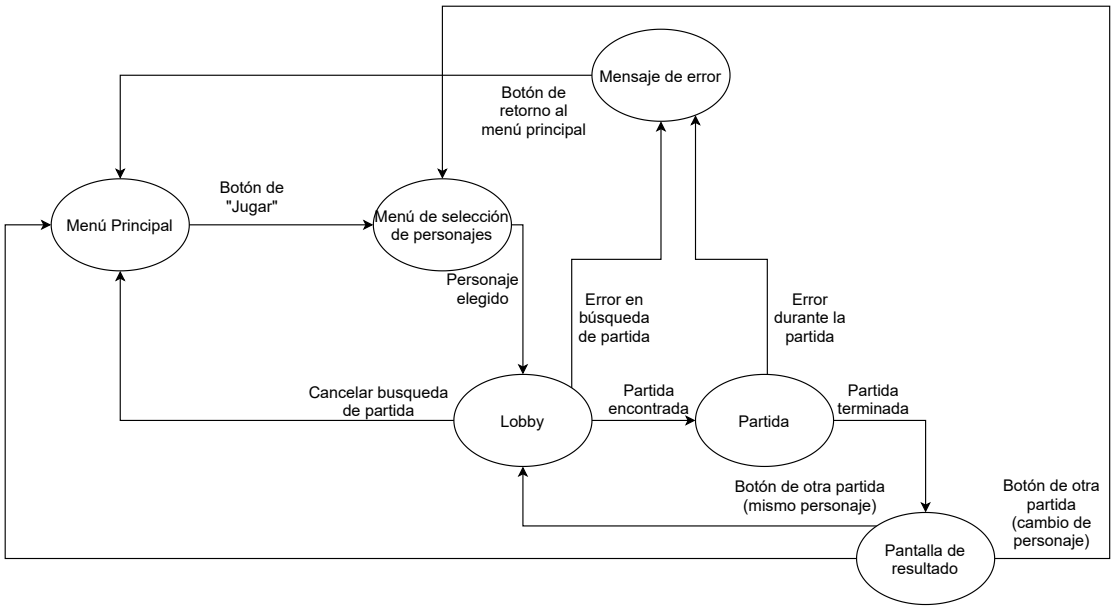

Figura 5.20: Gráfica de flujo de una partida

5.6.1. Inicio de sesión

Figura 5.21: Inicio de sesión

El juego comienza con un menú de inicio de sesión (fig. 5.21), que fuerza al jugador a crear o acceder a una cuenta (de forma que podamos guardar correctamente los datos de cada jugador).

5.6.2. Menú principal

Figura 5.22: Menú Principal

Una vez completado el inicio de sesión se muestra un menú principal (fig. 5.22), desde el cual se puede acceder a distintos menús: juego en línea, práctica, opciones, perfil de usuario e información del juego y contacto; además de ofrecer la opción de salir del juego.

5.6.3. Selección de personaje

Figura 5.23: Menú de selección de personajes

Al seleccionar esta opción se envía al jugador a un menú en el cual de le permite escoger el personaje a emplear en las partidas (fig. 5.23), con información relevante de cada uno. En caso de desearse, el usuario puede regresar al menú principal.

5.6.4. *Lobby*

Figura 5.24: Lobby previo a la partida

Una vez seleccionado el personaje, se accede a una pantalla de espera (fig. 5.24) mientras se busque partida, siempre permitiendo cancelar la búsqueda (mientras no se haya encontrado una).

5.6.5. Pantalla de resultados

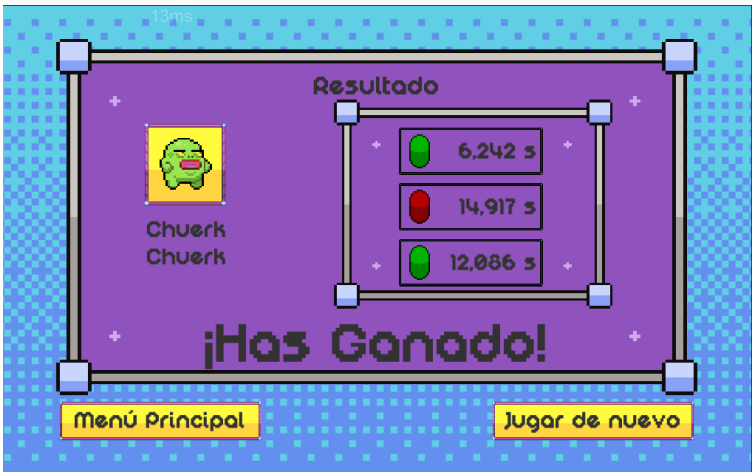

Figura 5.25: Pantalla de resultados

Transcurrida la partida, se muestra una pantalla con los resultados de la partida (fig. 5.25). Desde este menú se permite tanto regresar al menú principal como acceder directamente al *lobby* (buscando una nueva partida con el mismo personaje) como al menú de selección de personaje.

5.6.6. Menú de opciones

Figura 5.26: Menú de opciones del juego

Permite ajustar y configurar los siguientes parámetros:

- Pantalla completa: Sí o no
- Resolución de pantalla: Se recogen aquellas con las que el monitor en el que se ejecuta el juego es compatible
- Sincronización vertical: Sí o no
- Límite de fotogramas por segundo: 60, 75, 120, 144

- Volumen de la música del juego: Entre 0 y 100
- Volumen de los efectos de sonido: Entre 0 y 100
- Esquema de control: Teclado y ratón, o mando
- Idioma de juego: Inglés o castellano

5.6.7. Perfil de usuario

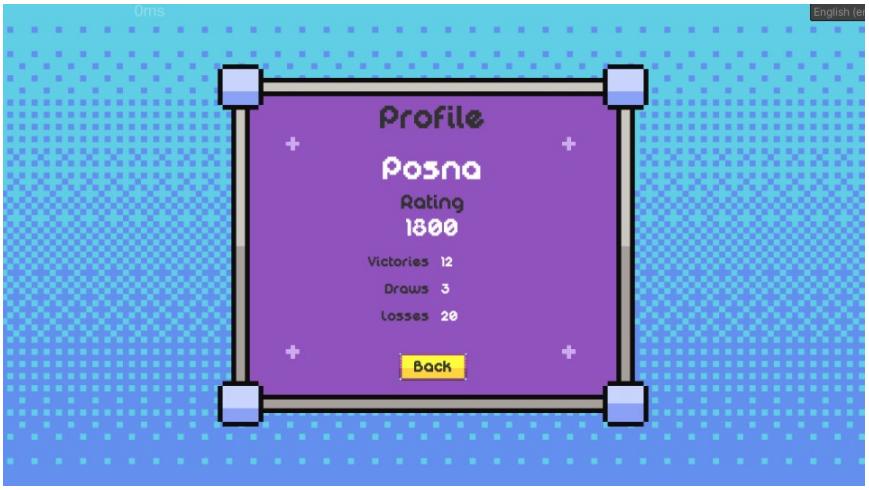

Figura 5.27: Menú de perfil de usuario

Permite al usuario ver su historial de partidas y su rendimiento (fig. 5.27).

5.6.8. Información

Figura 5.28: Menú de información del juego

Una pantalla con información (fig. 5.28) como:

- Información acerca del juego
- Redes sociales

- Opciones de contacto
- Créditos

5.7. Estética

5.7.1. Estilo artístico

Dada la leve experiencia del grupo en el apartado artístico, se ha optado por un estilo *pixelart* con el que estamos más familiarizados. Este estilo permite obtener un resultado agradable sin emplear una gran cantidad de recursos. Este estilo se emplea tanto en *sprites* bidimensionales (jugadores, interfaz, ataques) como elementos tridimensionales (que conforman el mapa).

Los personajes y las armas buscan ser simples y fácilmente reconocibles, con paletas coloridas y saturadas, colores brillantes. Sus diseños buscan dar la sensación de que los elementos del juego son como juguetes blanditos.

5.7.2. Música y sonido

La música y sonido buscan encajar con la temática *píxel* y retro del juego, empleando música animada y pertenecientes al género de 8 bits para rememorar el estilo de sonido de las máquinas arcade antiguas.

La música está dividida para cada una de las secciones del juego:

- **Menú Principal**: Música algo animada, corta y en bucle que trata de representar la fase previa al inicio de partida del juego.
- **Selección de Personaje**: Música previa que trata de representar la preparación para la batalla, más animada que la de menú principal.
- ***Lobby***: Música 8 bits más calmada para amenizar la búsqueda de contrincante.
- **Partida**: Música animada y rápida para representar el frenesí de una partida.

5.8. Conexión entre el juego y los servidores

El servidor de matchmaking ofrece una serie de servicios genéricos que requieren únicamente de un método de conexión a los servicios REST y clases que puedan deserializarse a objetos JSON para ser enviadas a través de mensajes.

En vez de hacer la integración directamente en el código del juego, la conexión a estos servidores se realiza mediante una librería dinámica (DLL) que hemos implementado para gestionar todo el proceso de envío de mensajes desde el juego hasta los servidores. Se ha creado con el fin de que otros desarrolladores puedan reutilizarla.

Así pues, el proceso final de conexión entre el juego y los distintos servidores se puede observar en la figura 5.29.

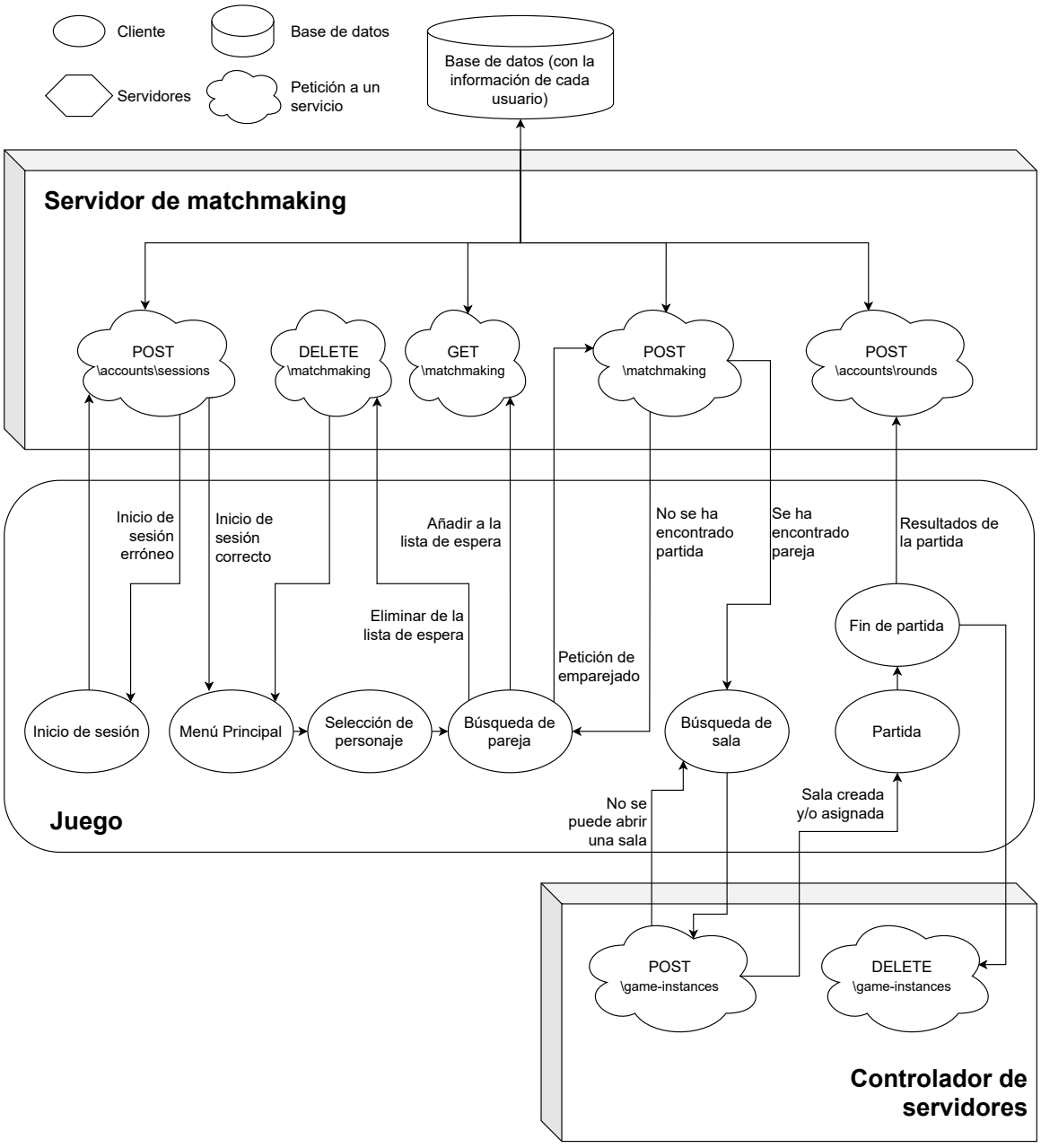

Figura 5.29: Esquema de flujo de conexiones realizadas durante una sesión de juego.

5.8.1. Gestión de cuentas

El inicio de sesión se realiza mediante una petición al servidor, al igual que la creación de nuevas cuentas. El inicio de sesión otorga un *token* de acceso que se empleará a lo largo de las distintas peticiones al servidor de *matchmaking*.

5.8.2. Emparejado

Este proceso, ya detallado en el capítulo 3 *Diseño del sistema de matchmaking*, se encarga de asignar al jugador un rival y devolver su identificador y nombre de usuario.

5.8.3. Servidor de juego

Con el fin de hacer los diferentes servicios independientes unos de otros, se ha implementado una arquitectura de Cliente-Servidor (el cual alojamos desde uno de nuestros equipos). Mediante el uso de la herramienta Mirror[4] se sincronizan las partidas de los clientes (jugadores), enviando mensajes al servidor con cada acción tomada por un usuario (movimiento, disparo, golpe de bala, etcétera) y replicándola en el cliente del otro usuario.

Mirror sincroniza las posiciones de todos los objetos que se instancian desde el servidor y cuentan con un componente llamado *Network Identity* y *Network Transform* como mínimo. *Network Identity* controla la identidad única del objeto del juego en la red y la usa para que esta reconozca al objeto. Este componente es imprescindible, ya que sin él no podría haber sincronización de otros datos. *Network Transform* sincroniza la posición, rotación y escala a través de la conexión con el servidor. Como estos componentes, existen otros para la sincronización de animaciones o posición de los hijos del objeto entre otros. Todos estos componentes están detallados en al documentación de Mirror[5].

Para que las funciones se ejecuten en el cliente o en el servidor, se emplean una serie de atributos que especifican desde dónde se llama a la función y dónde se ejecuta. Algunos de los atributos mas importantes son:

- [**Client**]: Especifica que una función determinada únicamente debe ser ejecutada en el cliente.
- [**Server**]: Especifica que una función determinada únicamente debe ser ejecutada en el servidor.
- [**ClientRpc**]: Se llama a la función desde el servidor pero la ejecuta en todos los clientes conectados.
- [**Command**]: Se llama a la función desde el cliente pero se ejecuta en el servidor conectado.

Estos atributos deben añadirse justo encima de métodos cuya clase herede de *Network Behaviour*.

Por otro lado, la sincronización de las variables debe ser especificada para asegurar que el estado de juego de ambos clientes sea el mismo y los dos usuarios dispongan de la misma

[4]https://mirror-networking.com/
[5]https://mirror-networking.gitbook.io/docs

información en pantalla.

Para señalizar la sincronización de variables, se ha de especificar el siguiente atributo:

- **[SyncVar]**: La variable indicada se sincroniza en los demás clientes solo si se modifica en el servidor.

Al igual que sucedía con los componentes, todos estos atributos se encuentran explicados y detallados en la documentación de Mirror[6].

5.8.4. Controlador de servidores

Se ha implementado un servidor independiente que gestione la creación de diversas salas de juego (o instancias de servidor de juego) en las que puedan jugar los dos usuarios emparejados, de forma que se puedan tener múltiples partidas concurrentes llegando a un máximo. Se realiza empleando un servicio REST con peticiones para abrir y cerrar una instancia de servidor de juego.

Servicios del controlador de servidores

Se han creado dos servicios en el mismo *endpoint*, */game-instances*, que permiten gestionar la creación o eliminación de un servidor de juego.

- **POST** */game-instances*: crea una nueva instancia del servidor de juego. Tanto si ya existía la partida como si la acaba de crear, devuelve el puerto en el que atienden las peticiones del juego.
  - *Respuestas*:
    - *200*: petición completada correctamente. La respuesta consiste en enviar el puerto en el que el jugador se va a conectar para empezar la partida. Formato: { port: int, matchID: string }
      - *port*: puerto en el que se debe establecer la conexión con el servidor de juego creado previamente.
      - *matchID*: ID de partida calculado a partir del ID de los jugadores y de la hora de instanciación del servidor.
- **DELETE** */game-instances*: marca como libre el puerto mandado desde el cliente.
  - *Respuestas*:
    - *503*: servidor lleno. La respuesta consiste en enviar únicamente el código de error. Indica que el servidor está al máximo de su capacidad y no admite más partidas.
    - *200*: petición completada correctamente. La respuesta consiste en enviar el puerto en el que el jugador se va a conectar para empezar la partida.

[6] https://mirror-networking.gitbook.io/docs/guides/attributes

Proceso de apertura y cierre de servidores

La apertura de un servidor, descrita en la figura 5.30, se realiza cuando el cliente recibe confirmación de que su usuario ha sido emparejado de forma mutua con otro jugador. En ese momento, cada uno realiza su propia petición al servicio POST, y envía su identificador y el de su rival. Una vez recibida esta petición, el controlador de servidores generará una cadena con estos dos identificadores, ordenándolos de menor a mayor y concatenándolos de forma que ambas peticiones produzcan la misma. Comprobará esta *key* en un diccionario, en el cual el controlador guarda las partidas en curso. Aquí pueden darse tres casos:

- Aún no existe una entrada: crea una instancia de servidor de juego con un puerto propio, tomado de una lista y genera un identificador de partida, el cual anota junto al puerto de conexión asociado con la instancia en el diccionario. También incrementa un contador que controla el número de partidas concurrentes en curso.

- Aún no existe una entrada, pero se ha alcanzado el máximo de partidas concurrentes: envía un error avisando de que no se pueden generar más partidas, avisando al cliente de que debe esperar a que haya una sala libre.

- Existe la entrada: se toma el puerto y el identificador de partida del diccionario.

En los dos casos en que se ha podido crear y/o asignar una partida, se envía el puerto y el identificador de la misma.

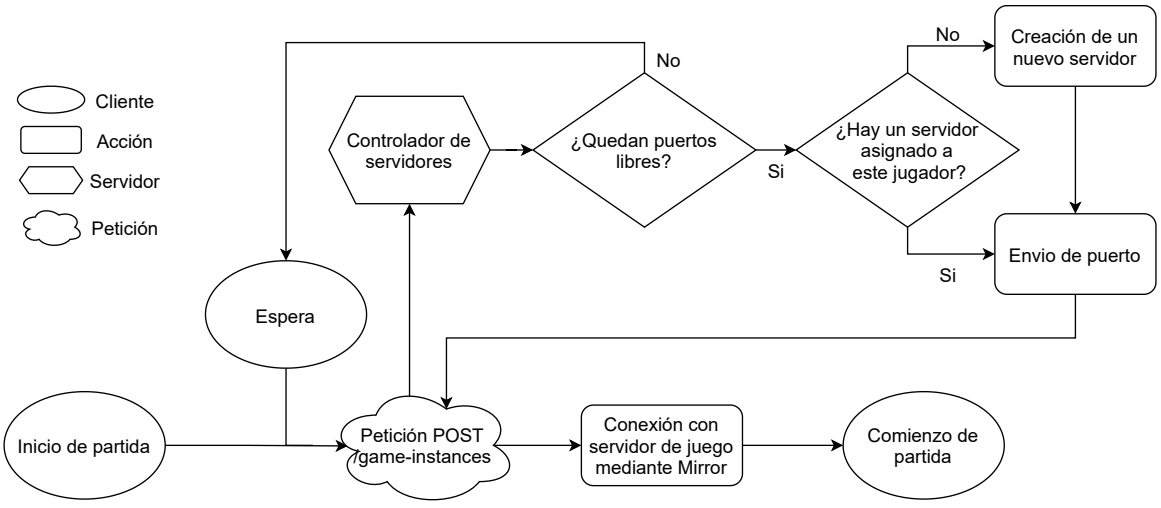

Figura 5.30: Esquema de conexión con el servidor de juego

El servidor creado finaliza su ejecución cuando la partida termina, ya sea de forma natural o por la desconexión de alguno de los dos jugadores, como se detalla en la figura 5.31. En ambos casos, ambos jugadores realizarán su propia petición al servicio DELETE con su identificador y el de su rival. Nuevamente, se generará la *key* del diccionario y se eliminará su entrada en caso de existir. Se debe realizar esta petición por parte de los dos jugadores dado que no tienen constancia de si el otro jugador ya ha realizado la petición.

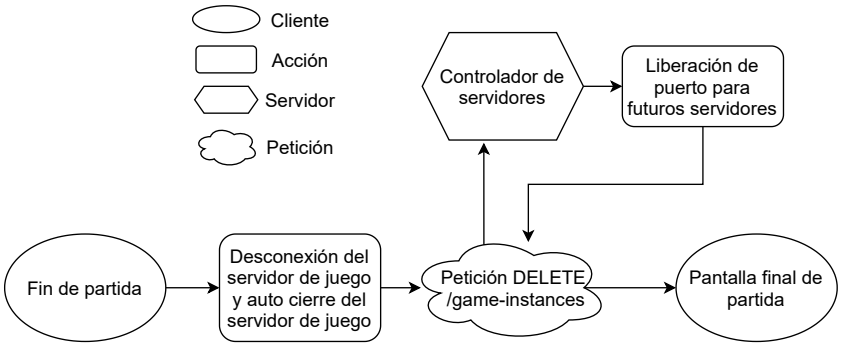

Figura 5.31: Esquema de desconexión con el servidor de juego

5.8.5. Fin de partida

Finalizada la partida, el cliente envía información recogida acerca de la misma mediante una petición al servidor de *matchmaking* (detallada en la sección 4.2.2 */accounts* como **POST** */accounts/rounds*) que lo añadirá en la base de datos al historial de partidas pendientes del jugador. Cada cliente envía la información pertinente a su jugador, y puesto que cada cliente requiere autenticación previa mediante un *token* (como se detalla en la sección 4.2.1 *Definiciones*), no es posible falsearla.

Esta información enviada siempre incluye el resultado de las tres rondas, y adicionalmente otros datos no necesarios para el cálculo de las puntuaciones. Estos datos se han decidido recoger de cara a facilitar análisis futuros del juego y su equilibrado. Incluyen detalles como la puntuación y desviación tanto del jugador como del oponente, e información de telemetría del rendimiento del jugador en la partida. Gracias al diseño del sistema, se pueden enviar todos estos datos adicionales (o no) sin necesidad de modificar la implementación del servidor para acomodarlo, ya que lo único estrictamente necesario es el resultado de las rondas.

En caso de desconexión de un usuario, se enviará la información tal y como se haría en una partida normal y considerando la partida como una total derrota por parte del usuario que ha abandonado la partida.

5.9. Herramientas

El desarrollo del juego ha empleado las siguientes herramientas:

- **Arte**
  - Aseprite
    - Software para creación de *sprites* estilo *pixel art*.
- **Motor de juego**
  - Unity (C#)
    - Motor de videojuegos gratuito de uso general, que admite creación de proyectos 2D y 3D.
- ***Plugins* de Unity**

- Mirror
  - *Plugin* creado para simplificar la gestión de un juego *online*, facilitando la creación de clientes y servidor de juego y la conexión entre los mismos.
- FMOD Studio
  - Sistema de efectos de sonido que facilita la gestión de eventos dinámicos de sonido, con un funcionamiento basado en eventos que se integran en el juego.
- 2D Tilemap Editor
  - *Plugin* nativo a Unity que permite crear mapas de *tiles* tanto en 2D como en 3D, empleando objetos tridimensionales.
- ProBuilder
  - *Plugin* empleado para crear los modelos de los objetos 3D para la construcción de los niveles.

Capítulo 6

Prueba con usuarios

Una vez finalizado el desarrollo del caso de estudio y su integración con el sistema de *matchmaking* (detallada previamente en el capítulo 4 *Implementación*), es necesario probarlo en un entorno con usuarios reales para comprobar su correcto funcionamiento. Así pues, en este capítulo se detallarán los objetivos de esta prueba, el procedimiento llevado a cabo para realizarla, y se analizarán los datos recogidos con el fin de extraer conclusiones.

6.1. Objetivos

El objetivo de esta prueba no es solo comprobar que el juego y los distintos servidores funcionan correctamente de cara al público, sino que también corresponde analizar la evolución de las puntuaciones de los jugadores a lo largo del periodo que mantengamos los servidores abiertos.

Las principales dudas son:

- En juegos competitivos es frecuente que se definan una serie de estratos de clasificaciones de forma natural, a medida que los jugadores juegan y evolucionan sus puntuaciones y desviaciones. ¿Se definen estratos de una forma visible?
- ¿Cómo se han realizado los emparejamientos a lo largo de las pruebas? ¿Qué diferencia ha habido entre las puntuaciones en estos emparejamientos? ¿Han ido evolucionando hacia enfrentamientos más precisos?
- ¿Cómo han evolucionado las puntuaciones a lo largo de las pruebas? ¿Ha sido un proceso gradual, o han cambiado de forma errática?
- ¿Qué diferencia existe entre usuarios que hayan jugado muchas partidas durante las pruebas y jugadores que hayan jugado menos?

6.2. Metodología

Para estudiar estos objetivos, se han definido la siguiente lista de pasos:

- **Establecer un canal de comunicación**: Es necesario un canal de comunicación para difundir información, llamar la atención de jugadores interesados y tener un

mejor control tanto de los reportes de posibles errores que puedan surgir como de la opinión general de y la comunicación con el público. Para ello se abrirá un servidor dentro de la aplicación de mensajería *Discord*[1]. Es una herramienta popular, que permite definir diferentes espacios de comunicación independientes y que no presenta límites de capacidad o actividad. Estos canales de comunicación serán los siguientes:

- **#avisos**: Esta sección servirá para comunicar avisos del comienzo y fin de las pruebas, además de anunciar situaciones derivadas de errores que puedan haber surgido durante las mismas.
- **#errores-y-problemas**: Esta sección servirá para que los jugadores puedan reportar posibles errores que surjan.
- **#general-offtopic**: Esta sección servirá para que los usuarios puedan hablar de forma más informal, con el objetivo de redirigir discusiones no relacionadas con los otros canales.

También contará con 5 canales de voz, en los cuales la gente podrá realizar llamadas de voz y compartir en directo sus partidas. Este servidor se mantendrá abierto hasta el final de las pruebas.

- **Buscar jugadores**: Es necesario contar con una base de jugadores lo suficientemente amplia y variada para realizar el análisis de forma correcta. Así pues, se difundirá el juego y el servidor de contacto por todos los canales disponibles: redes sociales, otros servidores de mensajería, etcétera.
- **Recoger los datos**: Una vez exista un público, se definirán unos periodos de tiempo para abrir los servidores y dejar que los usuarios jueguen. Así, se recopilará información de sus partidas que podrá posteriormente analizarse. Se prestará atención a cualquier reporte de errores, y se monitorizarán los servidores para detectar posibles fallos en los mismos.
- **Actualizar las clasificaciones con frecuencia**: Dado que tenemos un periodo corto de prueba y queremos probar todas las partes de este sistema, reduciremos el periodo de actualización a 15 minutos. De esta forma se tendrán más puntos de referencia a la hora de evaluar la evolución de las clasificaciones.

6.3. Proceso

6.3.1. Búsqueda de jugadores

El primer paso fue encontrar una base de jugadores que no solo tuvieran ganas de probar este juego sino que además sea lo suficientemente grande y variada para obtener datos que poder analizar de forma más completa. Se publicó el juego en Itch.io[2], un portal de distribución gratuito de videojuegos en formato digital, con el fin de tener una plataforma reputada de la que los usuarios puedan descargar el juego, así como atraer más público.

[1] https://discord.com/
[2] https://tefege.itch.io/tefege

Ya se realizaron una serie de encuestas (mencionadas en la sección 5.1 *Estudio de mercado*) para medir el interés por el juego entre otros detalles, y se ofrecía a los encuestados la posibilidad de compartir su correo electrónico con nosotros para poder avisarles a la hora de realizar pruebas como esta. Así pues, el primer canal de difusión del proyecto fueron estos correos electrónicos.

Aun así, era muy probable que estos encuestados no leyeran sus correos a tiempo, y muchos de ellos no proporcionaron correos electrónicos válidos. Por tanto, el segundo paso fue compartir los detalles de estas pruebas por los mismos canales que empleamos para las encuestas: grupos y chats de gente aficionada a los videojuegos, redes sociales, conocidos, etcétera.

En general, se espera que el público esté relativamente familiarizado con los videojuegos, pero que a la vez cuenten con un rango de habilidad amplio de cara a obtener un set de datos suficientemente variado al final de las pruebas. Se tiene en cuenta también que para la inmensa mayoría de usuarios será la primera toma de contacto con el juego, por lo que existirá un periodo inicial en el que los jugadores aprenderán las mecánicas del mismo.

6.3.2. Recogida de datos

El siguiente paso fue dejar que los jugadores pudieran probar el juego, recogiendo los datos tras cada partida completada. Al no poder dejar los servidores abiertos de forma indefinida, se definen los siguientes periodos de tiempo para las pruebas:

- Desde las 19:00 del sábado 12 de junio hasta las 00:00 del domingo 13 de junio.
- Desde las 18:00 del domingo 13 de junio hasta las 00:00 del lunes 14 de junio.

Este proceso fue relativamente automático, salvo por varias instancias en las que fue necesario realizar arreglos a los servidores de juego, al propio cliente de juego o a los servidores de *matchmaking* y actualización.

6.3.3. Actualizar las clasificaciones con mayor frecuencia

Para poder probar el proceso de actualización y ver su efecto a lo largo de los periodos de prueba, se redujo el tiempo entre actualizaciones de 1 hora a 15 minutos. Este cambio se realizó bajo la premisa de que los usuarios necesitarían un tiempo considerable de adaptación al juego, que no se podía ofrecer con los periodos establecidos para las pruebas.

6.4. Resultados

Por culpa de problemas técnicos ajenos a todos los sistemas desarrollados, fue necesario cancelar la primera sesión de pruebas y eliminar los datos recogidos durante la misma. Esto se debe a que los puertos especificados para los servidores de juego no tenían activados el reenvío de datos, por lo que las partidas se congelaban y sus resultados resultaban inválidos. Por tanto, únicamente se completó de forma satisfactoria la segunda sesión de recogida de datos.

Finalmente, se contó con 45 jugadores distintos, de los cuales 43 disputaron un total de 481 partidas. Así pues, de los datos recogidos de estos 43 jugadores se desprende el siguiente análisis. Tal y como se menciona en la sección 3.4 *Cálculo de la habilidad*, los jugadores comenzaron con una puntuación con valor de 1500 y la desviación fijada por defecto en 350.

6.4.1. Flujo de jugadores

Es importante tener en cuenta que el flujo de jugadores conectados en todo momento (figura 6.1, azul) no fue homogéneo, dando por ejemplo lugar a situaciones en las que dos jugadores se enfrentaban una y otra vez al no haber más rivales disponibles. Además, en la misma figura se puede observar el número de jugadores nuevos en cada periodo (figura 6.1, naranja), lo cual ofrece una idea más concreta del flujo real de jugadores a lo largo del tiempo, dado que estos introducen un alto grado de excentricidad al analizar valores por tener las puntuaciones por defecto.

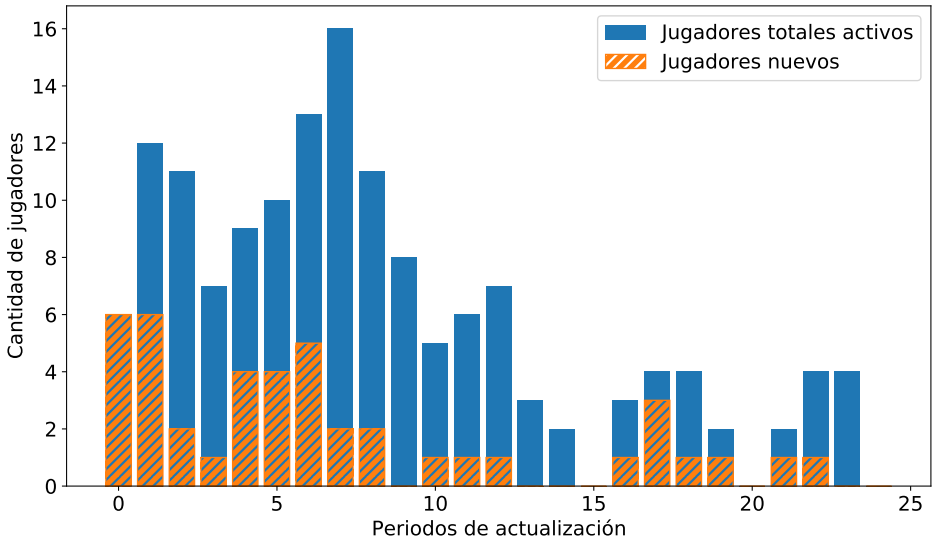

Figura 6.1: Medición del flujo de jugadores a lo largo de la prueba. En azul, el total de jugadores activos en un periodo de actualización. En naranja, la cantidad de estidos jugadores activos que son nuevos.

6.4.2. Estratos de clasificación

En la figura 6.2 se detalla la distribución de jugadores por puntuación, incluyendo la media y la mediana. Como se puede observar, existen algunos jugadores cuya puntuación se desvía enormemente, con el añadido de que son negativas. En ningún caso se menciona esta situación como algo imposible en la documentación de Glicko, llevándonos a asumir que son valores válidos del sistema. Dado que se realizan las actualizaciones de las puntuaciones por periodos, existe la posibilidad de acumular derrotas sucesivas frente a jugadores de un nivel de habilidad muy inferior al del usuario, y de tratarse además de un jugador con desviación elevada resultaría en un gran descenso de la puntuación.

Aun así, sigue resultando sorprendente ya que se han dado únicamente dos casos, de los cuales uno se encuentra en el rango de -9000. Si bien se ignorarán estos jugadores

a lo largo de análisis posteriores, dado que son una excepción, se detallará un análisis específico de estos casos en la sección 6.4.6 *Puntuaciones negativas*.

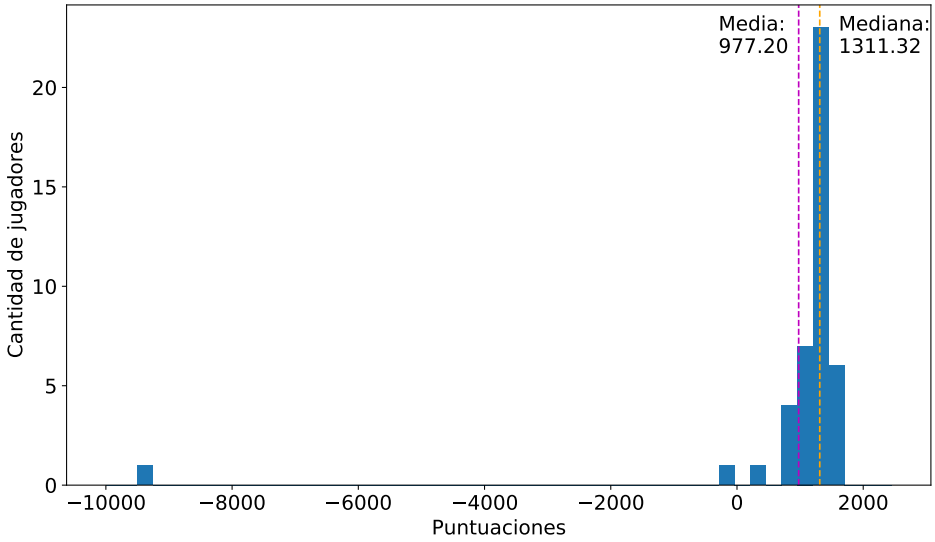

Figura 6.2: Distribución de los jugadores según su puntuación tras las pruebas.

Para hacer más cómoda la lectura, se ha creado la figura 6.3. Esta es idéntica a la figura 6.2, salvo que aparta visualmente estos casos excéntricos para mayor claridad.

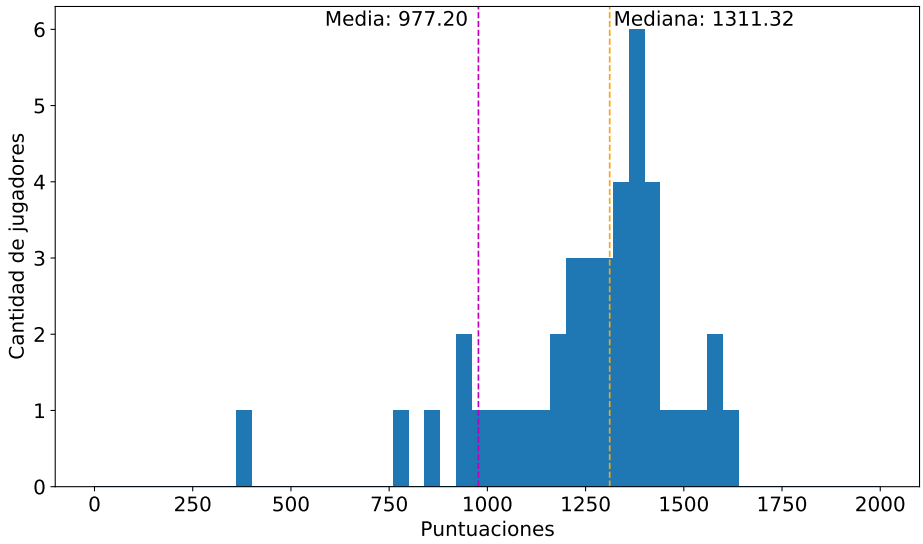

Figura 6.3: Distribución de los jugadores según su puntuación tras las pruebas omitiendo casos excéntricos.

Como se puede observar, la mayoría de puntuaciones (un 58,1 % del total) se encuentran entre 1200 y 1500. Si bien la cantidad de jugadores y de tiempo que han tenido disponible para aprender las mecánicas y jugar no es suficiente, pueden empezar a verse dos estratos definidos en la figura 6.4, que representa una versión de las dos figuras anteriores (figuras 6.2 y 6.3) enfocada en la zona entre 950 y 1500. Estos dos estratos son dos zonas de alta concentración de puntuaciones, y se encuentran:

- Entre 1200 y 1300.
- Entre 1350 y 1450.

También puede observarse el comienzo de un estrato en torno a una puntuación de 1000, y otro en torno a una puntuación de 1500. Este último caso es más llamativo, dado que 1500 es la puntuación inicial, significando que podrían definirse estratos superiores a este valor.

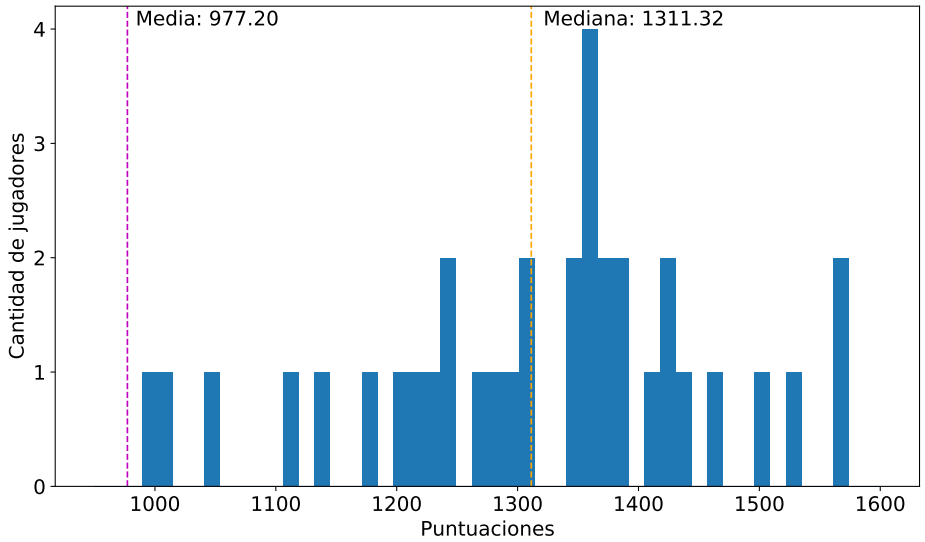

Figura 6.4: Distribución de los jugadores según su puntuación centrada en la sección de más concentración.

Aun así, se recalca nuevamente que esto ha sido una tendencia inicial, y dados más tiempo de pruebas y más jugadores los estratos se definirían de una forma más exacta.

6.4.3. Evolución de desviación

Tras el análisis de la puntuación y el surgimiento de estratos, es también necesario analizar el resto de elementos que intervienen en el sistema de *matchmaking*, principalmente la desviación.

Como puede observarse en la figura 6.5, partiendo del valor inicial de 350 las desviaciones finales descienden según lo esperado para situarse en un valor inferior a 100 en la inmensa mayoría de casos, un $83,7\,\%$ del total. Así mismo, existe también una base reseñable de jugadores (un $60,4\,\%$ del total) con una desviación en torno a los 50 puntos. La documentación de Glicko menciona este valor como el punto en el cual las puntuaciones de un jugador se han estabilizado, representando de manera más fiel su habilidad real.

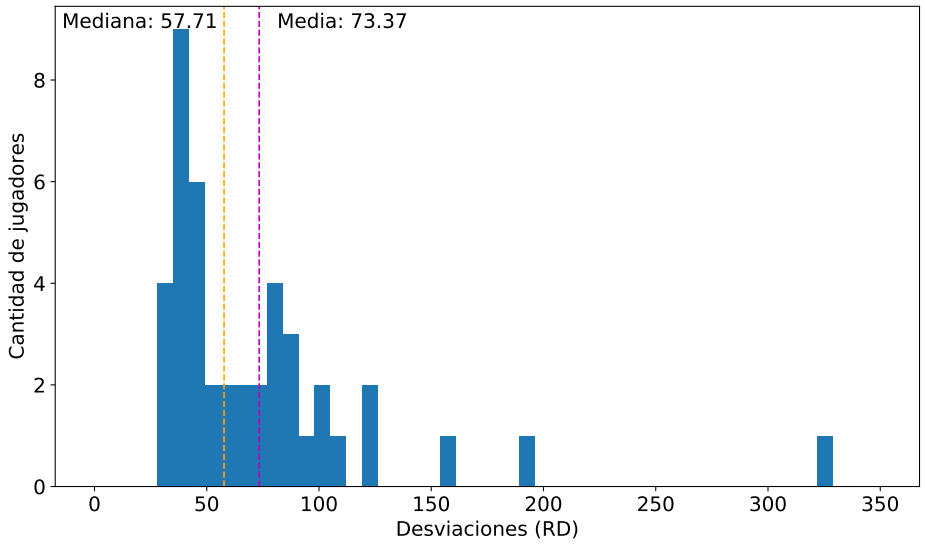

Figura 6.5: Distribución de los jugadores según su desviación tras las pruebas.

Dado el funcionamiento del propio sistema, es razonable estudiar la dispersión de las desviaciones en función del número de partidas jugadas, tal y como se refleja en la figura 6.6.

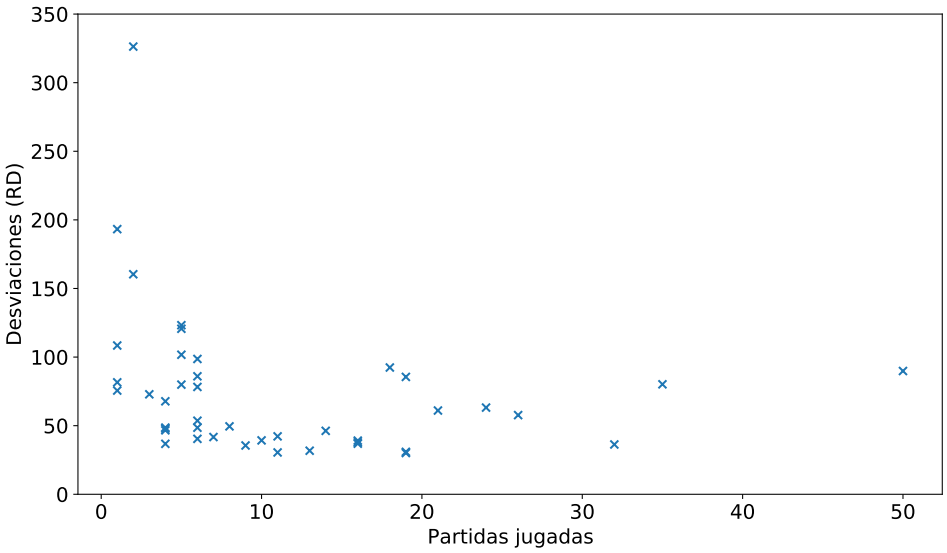

Figura 6.6: Nube de dispersión de la distribución de los jugadores, siendo el eje x el número de partidas jugadas y el eje y el valor de su desviación

Aunque en líneas generales se puede observar una estabilización notable según aumenta el número de partidas disputadas, es necesario tener en cuenta que la desviación aumenta en casos de inactividad, tal y como se mencionaba en la sección 2.2.2 *Glicko*. Esto podría dar lugar a casos que, con muchas partidas disputadas en el periodo inicial, un largo de periodo de inactividad, y un regreso en el periodo final, surgiera un valor distorsionado fruto del funcionamiento del propio sistema.

Sin embargo, tal y como se recoge en la ya mencionada figura 6.6, este caso no se ha dado en nuestras pruebas. Las desviaciones asociadas a jugadores que han participado en

menos partidas son más esporádicas, mientras que a medida que aumenta la cantidad de partidas jugadas las desviaciones se van concentrando.

Esta inestabilidad de desviaciones en torno a bajas partidas jugadas se debe principalmente a la baja cantidad de usuarios totales y alta proporción de jugadores recién llegados en los últimos periodos (figura 6.1, naranja). Dado que las puntuaciones y desviaciones de los oponentes entran en juego a la hora de actualizar los valores, una mayor proporción de jugadores nuevos provocaría que experimentaran un cambio más radical en sus desviaciones, sin poder estabilizarse.

Siendo la puntuación y la desviación de los jugadores los valores empleados para el *matchmaking*, es necesario también estudiar la relación entre ambos, que da lugar a la figura 6.7.

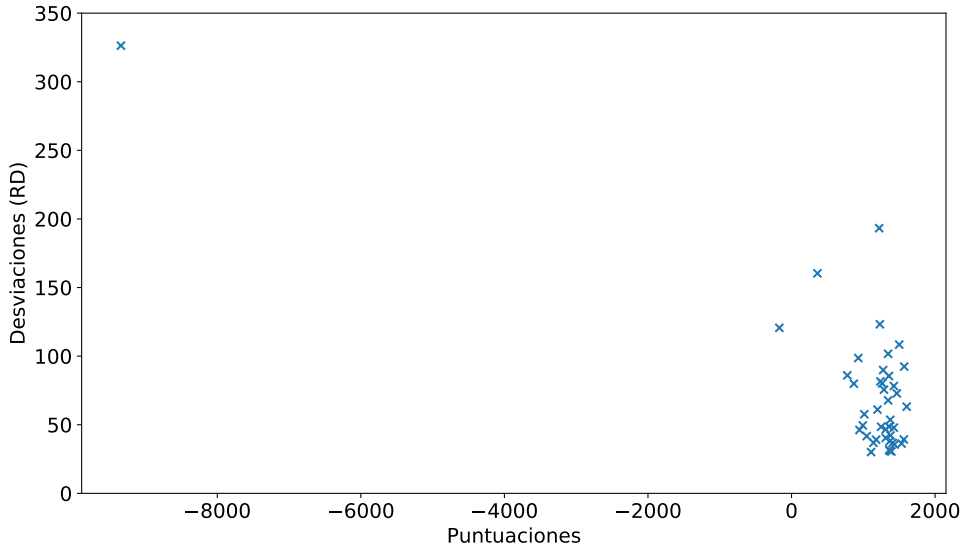

Figura 6.7: Nube de dispersión de la distribución de los jugadores, siendo el eje x el valor de su puntuación y el eje y el valor de su desviación

Al igual que con la figura 6.2, se pueden observar valores excéntricos fruto de emplear la puntuación como referencia (dada la existencia de valores negativos muy diferenciados del resto), por lo que se genera la figura 6.8 que los omite visualmente para una mayor claridad.

De aquí se desprende que los valores tienden a concentrarse con una desviación comprendida entre 25 y 100 independientemente de la puntuación. Así pues, dada esta independencia se puede afirmar que el sistema cumple con el objetivo de reflejar de forma fiable la habilidad de los jugadores por medio de sus puntuaciones.

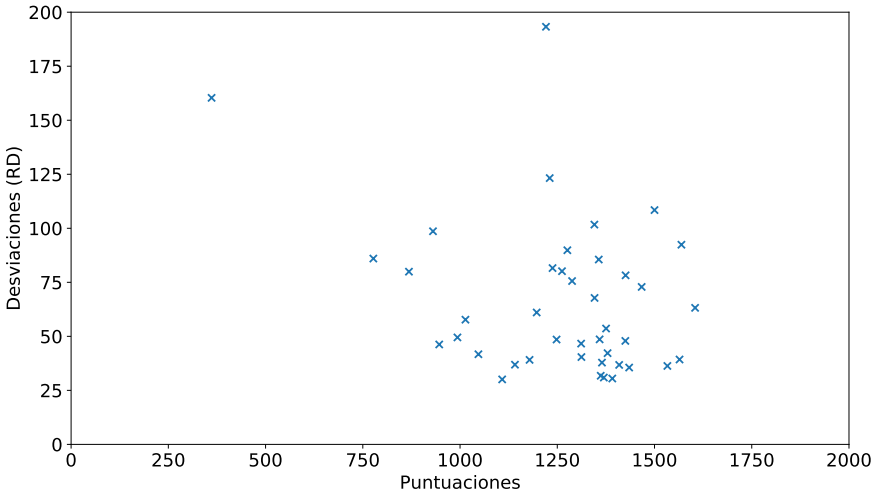

Figura 6.8: Nube de dispersión de la distribución de los jugadores omitiendo valores excéntricos.

6.4.4. Análisis de emparejamientos

Una vez estudiada la distribución tanto de puntuaciones como de desviaciones y su relación, se procede al siguiente objetivo de estas pruebas, siendo este analizar los emparejamientos realizados y su evolución a lo largo de las pruebas. Esto se realiza con el fin de tratar de discernir su calidad. El factor principal para poder afirmar esto es las diferencias entre puntuaciones al realizar los emparejamientos. Si descienden con el tiempo se podrá afirmar que los emparejamientos son más precisos. En este caso, el cálculo de habilidad habría logrado su objetivo de cara al emparejamiento, reflejando de manera más precisa la habilidad de los jugadores.

En la figura 6.9 se pueden ver representadas la dispersión de diferencias de puntuaciones en emparejamientos por cada periodo de actualización (azul) y la media de las mismas (morado).

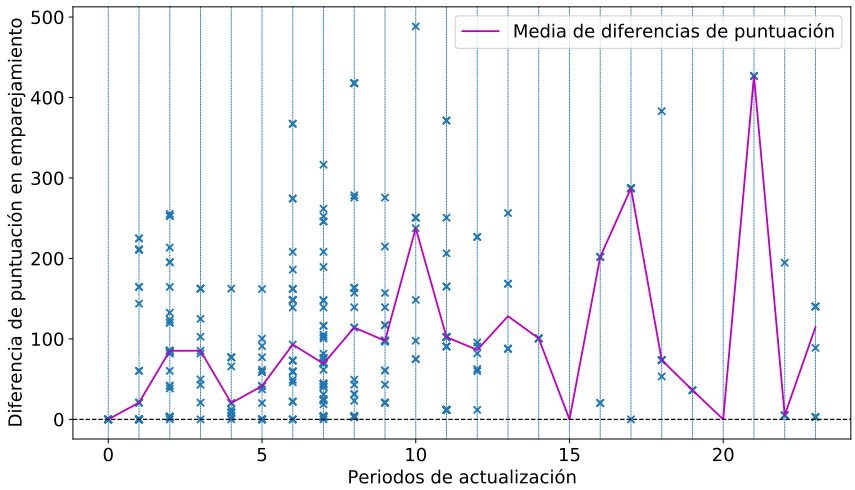

Figura 6.9: Distribución de la diferencia de puntuación de los emparejamientos. En azul la distribución de diferencias en un periodo, en morado la evolución de la media de estas.

En este caso se puede observar que, tras un primer periodo en el que todos los jugadores empiezan forzosamente con la misma puntuación, el valor fluctúa de manera estable con valores que se pueden considerar excéntricos en momentos con muy pocos jugadores conectados, fruto de la propia naturaleza de las pruebas al contar con un número limitado de usuarios en las mismas.

No se puede afirmar categóricamente que los emparejamientos se han realizado de manera equilibrada al no disponer de una muestra de datos más extensa con la que observar una evolución a lo largo del tiempo. Aun así, se puede observar que en los primeros periodos de las realizadas, con una mayor cantidad de partidas disputadas y menor proporción de jugadores nuevos (figura 6.1), la media de la diferencia de puntuación entre los jugadores en cada emparejamiento se mantiene estable. Si bien existen pequeñas variaciones, estas resultan aceptables, teniendo en cuenta la influencia de nuevos usuarios en este aspecto.

Al igual que ocurría anteriormente con el análisis del efecto de la desviación a la hora de realizar el cálculo de la habilidad, este es un apartado en el que la reducida cantidad de datos disponibles impide determinar un resultado de manera completa más allá de que su funcionamiento es correcto.

6.4.5. Evolución de puntuaciones y análisis individual

Por último, uno de los objetivos finales era analizar la evolución de las puntuaciones a lo largo de las pruebas, así como observar las evoluciones de jugadores que hubieran disputado muchas partidas, comparadas con las de jugadores que hubieran disputado pocas.

Debido a que el flujo de jugadores activos no ha sido constante a lo largo de los diferentes periodos de actualización sumado a la entrada de nuevos jugadores (figura 6.1), no sería lógico analizar estos valores sobre todo el conjunto de usuarios.

Es por esto que se realiza un análisis individual y comparación de distintos jugadores para estudiar los dos últimos objetivos de forma conjunta, ya que se encuentran íntimamente relacionados.

Para reflejar el contraste entre jugadores con alta y baja participación, se han analizado la puntuación y desviación de los tres jugadores con más partidas y de los tres con menos partidas (habiendo omitido aquellos usuarios cuya participación fuera nula o despreciable).

Participación

El primer paso es comparar la principal diferenciación entre estos dos grupos de jugadores, su participación a lo largo de los periodos de actualización.

Los jugadores con mayor participación (figura 6.10) son los usuarios 0 (35 partidas totales), 13 (32 partidas totales) y 21 (50 partidas totales). De la misma manera, los jugadores con menor participación seleccionados (figura 6.11) son los jugadores 4, 24 y 34, cada uno con 5 partidas totales. En sus respectivas gráficas se puede observar su participación a lo largo del tiempo.

En el caso de los jugadores de menor participación, resalta el hecho de que jugaron durante tres periodos como máximo. De manera similar, los jugadores con mayor participación tienen una inestabilidad parecida, jugando de forma irregular en vez de de una forma más distribuida.

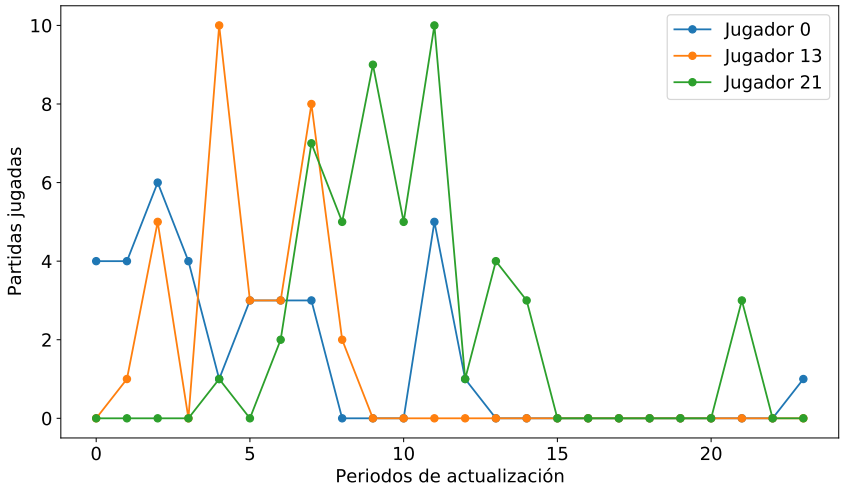

Figura 6.10: Progresión de las participación de los 3 jugadores con más partidas disputadas, representados por su identificador.

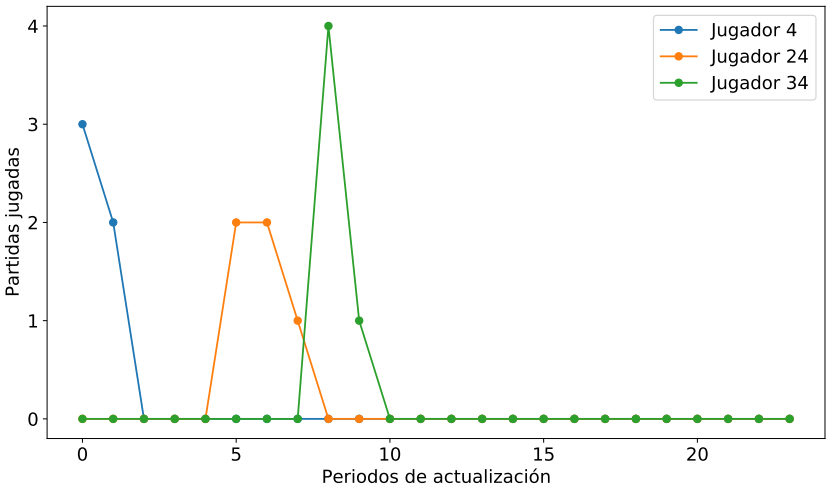

Figura 6.11: Progresión de la participación de los 3 jugadores con menos partidas disputadas escogidos, representados por su identificador.

Evolución de puntuaciones y desviaciones

Como puede observarse en la figura 6.12, la puntuación tiende a verse modificada de manera brusca en los primeros periodos de cada jugador (figura 6.10) para posteriormente estabilizarse este cambio. Esto se corresponde con la evolución de sus desviaciones como se describe en la figura 6.13, que como era de esperar se estabilizan según aumenta su participación. Cabe mencionar que este proceso no se corresponde con la cantidad de participación en cada periodo, sino que se estabiliza tras alrededor de 4 periodos de actividad.

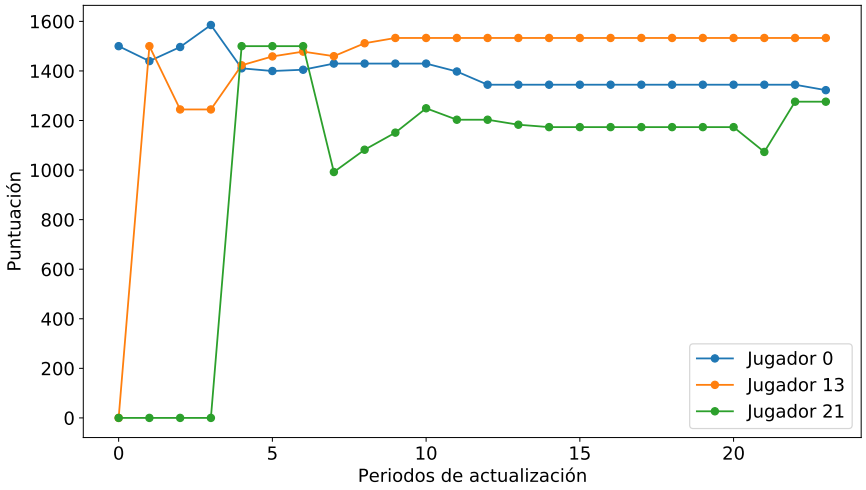

Figura 6.12: Progresión del nivel de puntuación de los 3 jugadores con más partidas disputadas, representados por su identificador.

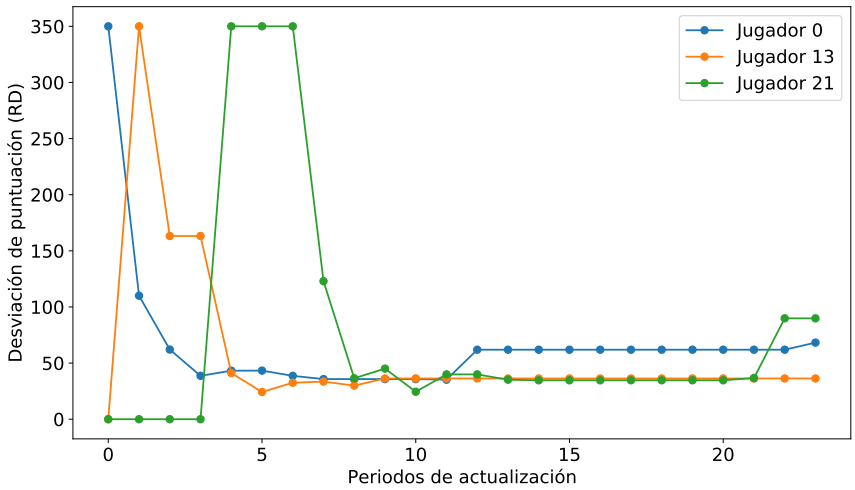

Figura 6.13: Progresión de las desviaciones de los 3 jugadores con más partidas disputadas, representados por su identificador.

Si bien ya se ejemplifica con los primeros periodos de los jugadores más activos, en el caso de los jugadores con menor participación se puede observar más claramente cómo las puntuaciones (figura 6.14) evolucionan de una forma más errática al no haber aportado suficientes datos al sistema y no haberse estabilizado sus desviaciones, como se puede observar en la figura 6.15.

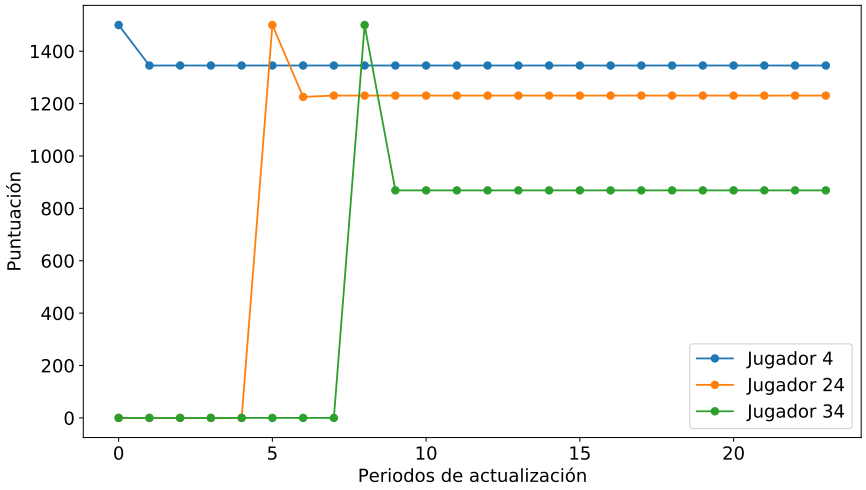

Figura 6.14: Progresión del nivel de puntuación de 3 de los jugadores con menos partidas disputadas, representados por su identificador.

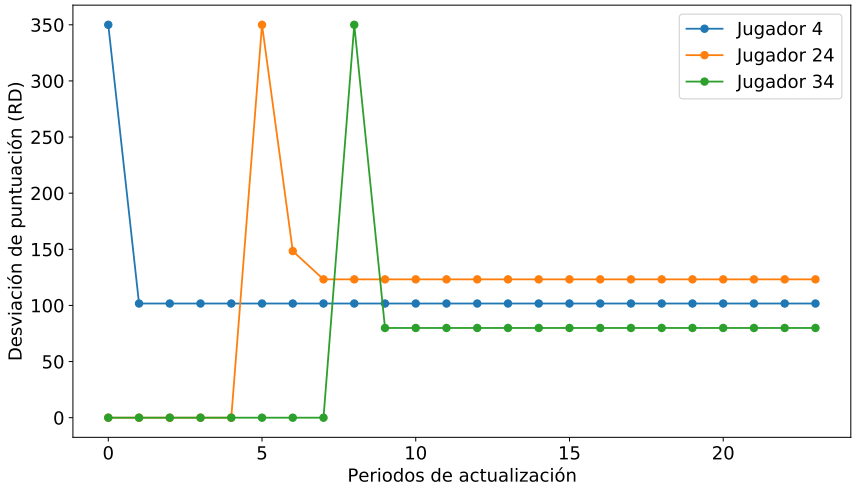

Figura 6.15: Progresión de las desviaciones de 3 de los jugadores con menos partidas disputadas.

A modo de conclusión, se puede afirmar que la puntuación evoluciona de manera errática al comienzo de la participación de un jugador, acorde a lo que se afirma en la explicación de Glicko (sección 2.2.2 *Glicko*). Esta evolución se suaviza progresivamente al descender la desviación, como se puede observar en la figura 6.12.

En relación al segundo objetivo, teniendo en cuenta que la media de partidas jugadas por usuario es de 11 partidas, puede afirmarse que una mayor participación implica valores mucho más estables (y evoluciones más suaves gracias a ello), mientras que en el caso contrario se pueden observar cambios más erráticos.

6.4.6. Puntuaciones negativas

Uno de los aspectos que surgió durante las pruebas fue la presencia de jugadores con puntuaciones negativas. Solo se da en 2 de los 43 jugadores (4,65 %) que participaron

en alguna partida, por lo que ha resultado llamativo a la hora de analizar el set de datos.

Como se ha aclarado previamente, no se menciona la imposibilidad de valores negativos en la documentación de Glicko (sección 2.2.2 *Glicko*) y pueden considerarse esperados en determinadas situaciones. Aun así, se ha considerado importante realizar un análisis de estos dos jugadores, con tal de discernir la razón de estos valores tan extremos y si podrían mitigarse modificando valores por defecto de Glicko.

Los dos casos

Para averiguar el por qué de estos valores, se ha realizado un análisis a fondo de los dos jugadores de puntuación negativa. Por fortuna, estos dos cuentan con pocas partidas, disputadas contra las mismas personas en un solo periodo, probablemente por el hecho de que la puntuación negativa impediría el emparejamiento. Así pues, los dos casos son:

- Jugador 38.
  - Puntuación: −170,253. Desviación: 120,645.
  - Partidas disputadas: 5.
  - Oponente: Jugador 7. Puntuación: 1212,746. Desviación: 103,283.
  - Resultados:
    1. [0, 1, 1]: Victoria.
    2. [1, 0, 0,5]: Empate.
    3. [0, 0, 0]: Derrota.
    4. [1, 0, 1]: Victoria.
    5. [0, 0, 0]: Derrota.
- Jugador 43
  - Puntuación −9342,36. Desviación: 326,304.
  - Partidas disputadas: 2.
  - Oponente: Jugador 21. Puntuación: 1073,186. Desviación: 36,689.
  - Resultados:
    1. [0, 0, 0]: Derrota.
    2. [0, 0, 0]: Derrota.

Tras aplicar los cálculos descritos en la sección 2.2.2 *Glicko*, aplicando los criterios descritos en la sección 3.4 *Cálculo de la habilidad*, se verifica que las puntuaciones eran correctas. Así pues, el siguiente paso es introducir cambios en el cálculo de las puntuaciones con el fin de observar cómo se modifican los resultados.

Cambio posible

Una duda persistente a lo largo del proceso de diseño del sistema fue la idea de las partidas con rondas, y cómo encajarían con el cálculo de la puntuación. En la sección 3.4 *Cálculo de la habilidad* se menciona la decisión de gestionar todas las partidas como realizadas por rondas, y a su vez la de gestionar los resultados de estas rondas de forma individual. En esencia, esto implica que todas las rondas contarían como una partida en sí, triplicando los resultados a tener en cuenta durante los cálculos. Esta decisión se tomó de cara a modificar Glicko lo mínimo posible.

Tomando estos dos jugadores de puntuación negativa, se ha realizado un nuevo cómputo de sus puntuaciones empleando como resultado de cada partida la media del de las tres rondas, con los siguientes resultados:

- Jugador 38.
  - Nueva puntuación: 150,447. Originalmente −170,253.
  - Nueva desviación: 187,833. Originalmente 120,645.
- Jugador 43
  - Nueva puntuación: −2459,448. Originalmente −9342,36.
  - Nueva desviación: 341,537. Originalmente 326,304.

Aun siendo un cálculo incompleto, puesto que los rivales no tendrían aplicados estos cambios, se puede observar una diferencia muy considerable entre los valores originales y los nuevos. Así pues, podría ser un punto de partida para mejorar el sistema de cara al futuro.

6.4.7. Conclusiones

El primer dato a mencionar es que se ha contado con una base de datos pequeña y con una referencia temporal muy limitada (incluso reduciendo el periodo de actualización para tener una muestra más completa en este sentido). Aun así, se puede extraer que tanto el sistema de clasificación como el propio emparejamiento funcionan de manera adecuada, cumpliendo con los objetivos marcados para estas pruebas y, en líneas generales, con lo propuesto en el trabajo a nivel global.

Incluso teniendo en cuenta situaciones excepcionales como la presencia de jugadores con puntuaciones negativas (que tampoco pueden llegar a ser consideradas un error), no se han detectado problemas en el funcionamiento del sistema. Aun así, el no poder desarrollarse estas pruebas de una forma más exhaustiva, muchas pruebas adicionales se han relegado a la sección 7.1 *Trabajo Futuro*, incluyendo la necesidad de realizar pruebas con más usuarios y durante un mayor periodo de tiempo.

Capítulo 7

Conclusiones y Trabajo Futuro

El objetivo del proyecto ha sido diseñar y desarrollar un sistema de *matchmaking* con una serie de restricciones, que pudiera implementarse para un videojuego multijugador en línea que se adaptara a estas. El sistema funciona como un servidor independiente del juego que tiene acceso a una base de datos de usuarios, y emplea esta información para responder a peticiones de búsqueda de partida emparejando los usuarios en línea de manera equilibrada, teniendo en cuenta el tiempo de espera que lleve un usuario buscando pareja, de forma que se amplíe la búsqueda. Así mismo, la actualización de clasificaciones se lleva a cabo en un servidor aparte, de acuerdo a su actuación en partidas previas.

Para ello, se ha realizado un estudio de múltiples sistemas de emparejamiento las estructuras empleadas, además de un estudio de diversos sistemas de clasificación. Se seleccionó Glicko debido a que sus características se adaptaban mejor a las características del proyecto. Aun así, el proyecto está diseñado de forma que, de desearse, pueda extenderse y cambiar el sistema de clasificación libremente, siempre y cuando se modifiquen las secciones relevantes del código.

Este sistema se ha desarrollado sobre un videojuego multijugador *online* para implementar el sistema de emparejamiento, y este juego ha sido utilizado como caso de estudio con usuarios reales para comprobar el funcionamiento del emparejamiento y la actualización de los parámetros involucrados en el sistema de *matchmaking*. También se ha desarrollado una librería dinámica que gestiona la conexión entre el servidor de *matchmaking* y el código en C# del juego.

En esencia, la librería es prácticamente independiente del juego, y podría emplearse en cualquier otro. Pero en el caso de que un desarrollador deseara cambiar la información que se recoge al final de una partida, esto requeriría editar una única clase de esta librería, aquella que es serializada a un objeto JSON para enviar los resultados de la partida a la base de datos, debido a las restricciones de C#.

Aun así, ofrecemos una versión adicional de la librería que se limita a enviar los datos estrictamente necesarios (resultado y duración de las rondas, id del oponente, id de partida) al servidor, y esta sí que podría ser aplicada de forma genérica a cualquier juego (siempre y cuando se ajuste a nuestros requisitos). También se ofrece en el servidor de emparejado la opción de definir una función que realice un procesado de los resultados de las partidas antes de subirlos al servidor, en caso de que se deseara realizar algún tipo

de análisis del rendimiento de los jugadores de forma previa.

7.1. Trabajo Futuro

Existen ciertas áreas de trabajo en las que se puede ampliar este proyecto, tanto de cara a realizar mejoras como en caso de necesitarle realizar pruebas más exhaustivas.

En primer lugar, uno de los desafíos más interesantes sería incluir la opción de realizar emparejamiento por equipos. Aunque es un proceso que ahora mismo podría hacerse de forma genérica (emparejando usuarios de forma individual y posteriormente agrupándolos en equipos), no tendría en cuenta posibles diferencias de nivel globales, usuarios que juegan juntos, o muchos otros factores importantes a la hora de gestionar equipos.

Es por esto que podría plantearse la inclusión de roles predefinidos o estilos de juego a la hora de realizar el emparejamiento: muchos juegos permiten definir (o definen ellos mismos mediante un algoritmo) el estilo de juego que más usa un jugador, ya sea en términos de clases (ayuda, ataque, defensa) o de estilos de juego (agresivo, defensivo, sigiloso, etcétera). Estos factores se tienen en cuenta a la hora de emparejar usuarios en equipos para que se complementen, o usuarios con estilos de juego opuestos para que compitan.

En segundo lugar, algo que podría llevarse a cabo es ajustar los diferentes parámetros que se tienen en cuenta para el cálculo de la puntuación (que ahora mismo son únicamente el resultado de la ronda o partida). En esta ocasión, con el fin de mantener el sistema genérico y adaptable a cualquier tipo de proyecto, no se han incluido más parámetros.

En este sentido, y aplicado de manera concreta al caso de estudio empleado en el trabajo, podrían incluirse parámetros como la precisión o el daño realizado, que son datos que actualmente ya se extraen de cada partida. Esta información podría ayudar a reflejar mejor la habilidad de un jugador y, enlazando con el punto anterior, podría facilitar también la generación de perfiles de usuario de cara al emparejamiento por equipos. Esto abriría la posibilidad de encontrar jugadores que se complementaran entre sí, haciendo la partida no solo justa en términos de habilidad sino también satisfactoria en caso de jugar con desconocidos. Otro parámetros interesante sería la duración de la partida o rondas, recompensando a los jugadores que se hicieran con la victoria de manera rápida y reduciendo el castigo de la derrota en caso de haber podido aguantar una mayor cantidad de tiempo.

Entrando en el plano más técnico de la implementación, y fruto de los resultados obtenidos en las pruebas con usuarios reales, se ha detectado que la vía empleada para computar los resultados (aplicando cada ronda de manera individual, en vez del resultado medio de las rondas de cada partida) genera cambios considerables en el cálculo tanto de la puntuación como de la desviación de los jugadores. De la misma manera, hemos detectado que la utilización de los valores recomendados por la documentación de Glicko presenta un margen de ajuste importante, especialmente en el ámbito de la desviación.

Dado que no podemos recalcular de manera fiable todas las puntuaciones en base a nuevos valores (ya que los emparejamientos generados podrían verse alterados, perdiendo la consistencia del sistema), si podemos señalar que sería interesante realizar nuevas sesiones de prueba aplicando estos cambios descritos.

De la misma manera, y en caso de generar una base de usuarios estable en cantidad y presencia a lo largo del tiempo, una posible vía de trabajo para el futuro sería la actualización de nuestro sistema base, siendo el más sencillo Glicko-2. Este es una versión extendida de Glicko que incluye la volatilidad de puntuaciones como parámetro, y no requiere ningún tipo de regeneración de puntuaciones al incluir un proceso de actualización desde valores de Glicko.

Por último, se planteó diseñar el sistema de manera independiente al juego que empleamos como caso de estudio. Para esto se ha desarrollado una librería dinámica, la cual gestiona la comunicación entre cliente y servidor. Mas, a la hora de subir información a la base de datos, C# requiere especificar un formato mientras que el servidor y la base de datos permiten introducir datos de formatos flexibles. Se ha desarrollado una versión de la librería que se limita a enviar los resultados mínimos necesarios, y otra que se adapta a las necesidades de nuestro caso de estudio. Cabe destacar que este problema de la librería dinámica es el único que nos impide tener un sistema completamente genérico para C#, y hemos podido probar la genericidad del mismo simulando conexiones desde otros lenguajes que sí permiten el envío de datos con un formato flexible.

Así pues, como trabajo futuro sería interesante investigar cómo solventar este hecho y probar que el sistema sea aplicable a otros juegos hechos en C# sin necesidad de alterar la librería, de forma que siga teniendo la flexibilidad que se planteó ofrecer desde el principio. Esto se aplicaría a juegos que sean similares o de géneros distintos, siempre y cuando compartan la restricción de que las partidas deben ser de uno contra uno.

Capítulo 8

Contribuciones

Pablo García Grossi:

Para comenzar con el planteamiento del trabajo y cómo debíamos organizarnos, decidí centrarme en la búsqueda de información acerca de los sistemas de medida de habilidad de jugadores en videojuegos en línea. Realicé una recopilación de los principales métodos de puntuación utilizados y los presenté al resto del grupo para analizar cuál sería el más apropiado para el desarrollo de nuestro proyecto.

Tras nuestra decisión de utilizar Glicko como sistema de puntuación, realicé una implementación previa en Python en la que, con una serie de jugadores generados en un documento JSON que constaban de un id, puntuación y desviación. Esta prueba general consistía en la selección de un usuario aleatorio de la lista y someterlo a 20 partidas con usuarios que ronden su rango de puntuación y, posteriormente, actualizar su puntuación. El objetivo era comprobar que Glicko era un sistema adecuado para el funcionamiento y diseño de nuestro juego.

Posteriormente participé en el diseño y desarrollo del videojuego. Por un lado, me encargué de todo el apartado de arte, en el que utilicé la herramienta Aseprite para la elaboración de los diseños de las armas, personajes, interfaces y elementos del entorno en el mapa. Durante este proceso, colaboré en la descripción y diseño del funcionamiento y habilidades de los personajes disponible.

Tras haber contribuido con el apartado artístico dentro del juego, colaboré en el desarrollo de las mecánicas principales del juego, ayudando en la implementación del movimiento, apuntado, disparo y lanzamiento y creación de habilidades. Me encargué del pulido del movimiento, estableciendo parámetros correctos y solucionando algunos problemas como el movimiento en diagonal. También me encargué del correcto apuntado y lanzamiento de balas, adaptando el código para su correcto funcionamiento respecto a nuestro estilo gráfico. También me encargué de la elaboración del sistema de habilidades junto a **Jose Martín Serrano** e **Ignacio Ory Alonso**, además de la implementación de gran parte de los distintos ataques y disparos de personajes.

Dejando de lado las contribuciones al apartado del juego, colaboré en el diseño y organización del sistema de *matchmaking* junto a mis compañeros, planteando la estructura fundamental de este y su funcionamiento. Mi principal labor en este apartado fue organizar la estructura y elaborar los esquemas para la implementación de los servidores,

además de organizar las conexiones entre ambos y la filosofía de diseño que fuésemos a adoptar.

Antes de ponerme con un sistema de pruebas para la comunicación cliente-servidor y el emparejamiento de usuarios, investigué y colaboré en la comunicación de Unity con MongoDB, así también como plantear el almacenado de usuarios en la base de datos y buscar e implementar el uso de cifrado SHA256 y funciones para comprobar la corrección de nombres de usuario, correos electrónicos y contraseñas.

Tras este planteamiento inicial, me encargué de desarrollar un sistema para poner a prueba las conexiones con el servidor de *matchmaking*, controlar las peticiones y poner a prueba su correcto funcionamiento con un sistema de usuarios ficticios similar al elaborado previamente en Python, pero añadiendo y teniendo en cuenta todos los nuevos parámetros a tener en cuenta en nuestro videojuego.

En este proceso me encargué de readaptar las pruebas previas en Python a Javascript para la permitir la correcta comunicación de Glicko con el servidor de *matchmaking*, además de implementar y establecer dentro de los cálculos de puntuación los nuevos parámetros que íbamos a tener en cuenta a la hora de evaluar el nivel de habilidad de los usuarios para demostrar la adaptabilidad de nuestro sistema a distintos videojuegos.

Durante las pruebas, colaboré con **Javier Arias González** para observar el correcto funcionamiento del servidor de tipo REST y junto a él corregimos todos los problemas surgidos durante las pruebas, estableciendo conexiones síncronas correctas, reestructurando ciertas clases serializables y solucionar todos los posibles problemas de comunicación entre cliente y servidor, además de comprobar la correcta implementación de las prácticas generales de nomenclatura de REST.

En esta fase de testeo, creé un cliente que tomaba control de la conexión y desconexión de jugadores y simulación de partidas. Su función principal era poner a prueba el correcto emparejamiento de jugadores, teniendo en cuenta su puntuación, desviación y tiempo de espera en la cola de jugadores.

El sistema se inicializaba poniendo a prueba el registro de jugadores, introduciendo en la base de datos todos y cada uno de los jugadores generados aleatoriamente en un JSON, incluyendo su nombre de usuario, correo electrónico, contraseña y una puntuación y desviación aleatorias. El registro se realizaba una única vez y a continuación el cliente seleccionaba a un total de 10 jugadores aleatorios que introduce en la cola de espera del servidor de *matchmaking*, emparejándolos entre sí y permitiendo comprobar el correcto funcionamiento del sistema. Este contaba el número de partidas que iba realizando cada uno de los jugadores y aleatoriamente simulaba una desconexión de los jugadores en línea, desconectándolos de la cola de espera del servidor de *matchmaking* e introduciendo nuevos jugadores a la espera de una partida, todo esto en bucle para poder confirmar la solidez del sistema, tanto el emparejamiento como el correcto funcionamiento entre la conexión cliente-servidor.

Jose Martín Serrano:

Como todos mis compañeros, el inicio del proyecto consistió en buscar información sobre el *matchmaking*, charlas y documentos en los cuales se hablase de como funcionaban y juegos que lo utilizaban. Aunque no entré en tanta profundidad como algunos de mis compañeros, comprendí el funcionamiento del Elo y Glicko, además de su uso en algunos

juegos.

Una parte principal de mi trabajo ha ido en la creación del videojuego. Sobre todo me centré en la estructura principal de los personajes y su mecánicas. La base de la mecánica de disparo de los distintos personajes ha sido una de mis mayores contribuciones a la construcción del videojuego, junto a todo el sistema de daño y vida para el correcto desarrollo de una partida. Aparte, la creación de una maquina de estados para poder añadir la mecánica de ralentización, enamoramiento, aturdimiento, etc... con el fin de poderlo extender para añadir distintos estados.

Dentro de la parte del juego, también me centré en pasar todo aquello que hicimos de forma local al servidor mediante el uso de la librería Mirror. Este proceso incluía la investigación del funcionamiento de la herramienta para poder verificar que la íbamos a poder usar para este proyecto, que comprendía mirar tutoriales, leer la documentación disponible y observar proyectos de ejemplo ya hechos (principalmente los ejemplos que aportaba Mirror por defecto). Tras verificar que la librería nos podía ser útil, me encargué de la implementación, pasando todo el funcionamiento que teníamos en local a un servidor de juego que manejase la conexión entre clientes y unos clientes los cuales se conectaban a este para poder jugar una partida 1v1.

Por último, relacionado con el videojuego, también me encargué de buscar errores y el posterior arreglo de estos. Algunos de estos problemas no eran solo de errores del videojuego en sí, si no también de como se comportaba el juego respecto los servidores (a la hora de procesar las rondas o casos de desincronización).

Sobre los servidores, al igual que mis compañeros, contribuí en el diseño de la jerarquía de servidores, así como el flujo de partida. Debido a que me encargué de la parte en línea del videojuego, investigué cómo funcionaban los servicios REST y qué software era necesario. Una vez encontrado información sobre Express (librería que utilizamos para los servidores en Javascript), hice un pequeño servidor de prueba que sirvió como plantilla para la extensión posterior del actual servidor de *matchmaking*.

El servidor de juego necesitaba poder alojar varias partidas a la vez para poder seguir con el *matchmaking* por lo que estuve investigando como podíamos crear diversas salas para que distintos clientes se pudieran conectar y pudiesen jugar partidas de forma separada (por parejas, dado que en nuestro caso de uso,los jugadores se enfrentaban entre sí de manera individual). Dentro del abanico de posibilidades y sabiendo como funcionaban los sistemas REST, la solución que encontré fue crear un servicio REST que crease y finalizase partidas a petición de los clientes. Este servidor crea una instancia de un servidor que escucha en un puerto concreto y envía a los clientes el puerto en el que escucha y el identificador de la partida para seguir la partida posteriormente.

Desde el lado del cliente, necesitábamos poder hacer peticiones desde C#, así que trabajé en la búsqueda de una forma que nos permitiese la comunicación desde el cliente hasta el servicio REST. Encontré HttpWebRequest y creé una clase en el proyecto con distintos métodos para poder establecer comunicación con el servicio REST y poder, entre otras posibilidades, iniciar sesión o buscar contrincante. Esta clase sufrió cambios tras modificar la forma en la que gestionamos las peticiones desde el servidor de *matchmaking*. Tras los cambios que sufrió esta clase, añadí funcionamiento para que los clientes pudieran comunicarse con el controlador de servidores de igual modo que lo hacían con el de *matchmaking*.

Esta clase se terminó transformando en la librería dinámica que usamos para la comunicación tanto con el servidor de *matchmaking* como con el controlador de servidores. Debido a que en los inicios no estaba pensada como una librería dinámica genérica para cualquier videojuego, me encargué de separar el código genérico y el específico para crear una librería dinámica genérica y otra más específica esencial para toda la comunicación del juego.

También necesitábamos recopilar algunos datos de los jugadores para guardarlos posteriormente en la base de datos externa que teníamos, así que me encargué de modificar código de la parte del juego para poder recoger estos datos, que encapsulamos posteriormente para enviárselos al servidor de *matchmaking* mediante la clase creada anteriormente exclusivamente para este tipo de comunicación.

De forma externa a lo que era el desarrollo del TFG, me ocupé de la creación del repositorio TFG en Github donde teníamos como submódulos todos los repositorios utilizados (repositorio de juego, repositorio de *matchmaking*, repositorio de actualización, repositorio de la comunicación de los clientes, repositorio de la comunicación con Mongo y repositorio del controlador de servidores de juego).

También me ocupé de redactar algunos de los Readme de los repositorios con algo de documentación necesaria para poder modificar el código de aquel que quiera.

Javier Arias González:

Al comienzo del proyecto los cuatro integrantes buscamos información acerca de sistemas de *matchmaking* en sí mismos, además de los métodos de conexión entre clientes que estos empleaban. Por mi parte lo que encontré fueron proyectos más antiguos, uno de ellos de cara a *matchmaking* en un sentido comercial (emparejar anuncios con potenciales clientes). Aun así, estos proyectos nos ayudaron a la hora de empezar a esquematizar lo que sería nuestro propio sistema, además de introducirnos a conceptos más técnicos de este campo.

También participé en el diseño original del juego, centrándome en el desarrollo de las mecánicas principales y su equilibrado, además de el aspecto estético (no los *assets*, los cuales en su mayoría cayeron en manos de mi compañero **Pablo García Grossi**). También me encargué de buscar *plugins* que nos simplificaran el trabajo, como a la hora de construir los mapas, y de integrar el proyecto realizado para la asignatura de **Sonido en videojuegos** con este mismo juego, consistente en añadir música y efectos de sonido dinámicos.

Quitando estas dos contribuciones iniciales, unas de mis contribuciones principales fueron los servidores de matchmaking, actualización y la integración de la base de datos MongoDB. El diseño general fue realizado por los 4, y la investigación inicial de la integración de REST en node.JS fue realizada por **Jose Martín Serrano**, que realizó un servidor de plantilla que luego emplearía a la hora de diseñar el servidor en su totalidad.

Al principio el servidor fue realizado iterativamente, a medida que se necesitaban servicios los iba diseñando, lo cual provocó que fuera necesario reestructurarlo de forma correcta. Peor aún, las conexiones con MongoDB requerían asincronía, dado que el *driver* de MongoDB trabajaba mediante promesas. Así pues, recayó sobre mí la tarea de lograr que se conectara de forma correcta y síncrona, además de ajustarme lo máximo posible a las prácticas generales de REST. Para empezar, creé una pequeña API que gestionara

la conexión con MongoDB, de forma que cambiar la base de datos de cara a posibles modificaciones fuera relativamente sencillo.

Siendo mi primera introducción a estos conceptos, requirió varios intentos aprender y organizarlo todo de forma adecuada, ajustando los servicios y la gestión de códigos de errores, pero al final logré terminarlo y probarlo con la ayuda de **Pablo García Grossi**, que había diseñado un sistema para realizar pruebas de estrés sobre los distintos servicios. También me encargué de meter varios niveles de seguridad configurables: emplear el servidor en *https*, y emplear un sistema de *tokens* de autenticación para verificar la identidad de un cliente sin necesidad de enviar los credenciales constantemente. Todo este trabajo culminó en los servidores actuales.

Creados los servidores, la siguiente tarea fue realizar la conexión entre Unity y estos. Esta tarea fue más complicada que la anterior, ya que C# no posee la flexibilidad de JavaScript, con la cual conté a la hora de montar el sistema de respuestas. Fue necesario crear clases que pudieran serializarse y deserializarse acorde a los datos recibidos de los servidores, además de gestionar los errores que pudieran surgir en forma de pantallas de error generadas por el propio juego.

Una vez implementado, fue imperativo probar estos sistemas en un ambiente más realista, por lo que **José Martín Serrano** se ofreció para ejecutar los servidores desde su propio ordenador de forma que pudiéramos jugarlo y probar que todo funcionaba de forma adecuada. Esto, lógicamente, sacó a la luz varios problemas que no habíamos tenido en cuenta, y si bien la mayoría se debían al propio juego algunos también recaían sobre los servidores. Así pues, dedicamos un par de tardes a solucionar y pulir estos errores para, al final, lograr que tanto el juego como los servidores funcionaran sin ningún tipo de problema.

Aparte de los servidores también me encargué del redactado de la memoria, siendo el más familiarizado con los sistemas internos de los servidores, además de ser el único integrante que no contaba con asignaturas optativas el segundo cuatrimestre (y por tanto, el que contaba con más tiempo libre). Esto incluye además estudiar LaTeX (y TeFlonX, la plantilla que estamos empleando ahora mismo) de forma que luego pudiera ayudar a mis compañeros con dudas de forma más rápida.

Por último, nuevamente debido a que este segundo cuatrimestre contaba con más tiempo libre que el resto, me dediqué a ofrecer ayuda a mis compañeros en sus respectivas tareas, además de revisar código y memoria en busca de posibles errores.

Ignacio Ory Alonso:

Siguiendo un orden cronólogico, como ya han comentado algunos de mis compañeros, empezamos este trabajo investigando y buscando todo tipo de información sobre sistemas de partidas multijugador, de forma genérica en un primer momento para luego profundizar en todo lo relativo al sistema de *matchmaking*. En este sentido, yo me he centrado principalmente en todo lo relativo al Estado del Arte, con diferentes técnicas y tendencias relacionadas con las partidas multijugador en diferentes juegos. Esto incluye técnicas para aprovechar el comportamiento psicológico de los jugadores en relación al proceso de emparejamiento (por ejemplo, que se vean recompensados incluso aun cuando pierden), así como elementos propiamente externos al propio sistema de emparejamiento, con métodos empleados para que la espera entre partidas se haga más amena, por ejem-

plo), así como respecto a los diferentes sistemas empleados en diferentes proyectos, tanto enfocados al emparejamiento individual (como Elo o Glicko) así como aquellos diseñados de forma expresa para emparejamientos por equipos.

A la par que llevábamos a cabo este proceso, íbamos creando también un juego original creado expresamente para ser empleado como prueba de uso para el sistema de *matchmaking* a desarrollar. Este ha sido uno de los elementos en los que más me he implicado a nivel de desarrollo como tal, tanto a nivel de diseño, incluyendo diferentes mecánicas, como los propios personaje, cada uno con un arma y habilidad diferente, así como la estructura del mapa para dar lugar al estilo de juego frenético que buscábamos. Pasando al terreno de la implementación, además de algunas de las habilidades previamente mencionadas, me he encargado del sistema de menús e interfaz, además de añadir un modo de práctica como herramienta tanto para que los jugadores pudieran probar los personajes en un entorno relajado, además de servirnos a nivel de desarrollo como vía para comprobar que todos los nuevos elementos funcionaban de la manera esperada (y poder balancear algunos aspectos de cara al modo multijugador).

Dentro del apartado de la interfaz, también he participado en pruebas con diferentes usuarios externos al desarrollo del proyecto de cara a asegurar que todos los elementos se percibían de la manera deseada (en términos de claridad y tamaño, para ser suficientemente visibles para transmitir la información necesaria, pero sin tampoco llegar a agobiar o molestar al jugador), con los consiguientes análisis de comentarios y opiniones para solventar algunos errores detectados.

De la misma forma, también me he encargado de la adaptación e inclusión de diferentes herramientas y *plugins*: Unity Localization para poder traducir y localizar el juego de forma prácticamente automática (de cara a poder ampliar la base de usuarios potencial que pudiera participar en la fase de pruebas) haciendo uso de tablas de equivalencia entre idiomas, FMOD para lo relativo al sistema de sonido, incluyendo la creación de eventos y bancos de sonidos con la herramienta complementaria FMOD Studio, así como la integración inicial de Mirror, *plugin* que hemos empleado para simplificar la gestión de servidores y clientes de Unity para todo lo relativo al correcto funcionamiento del juego.

Aunque no he llegado a entrar de manera tan profunda en la codificación los sistemas de servidores y *matchmaking* como sí han hecho algunos de mis compañeros, participé junto al resto en el diseño general de nuestro propio sistema de *matchmaking*, lo cual incluía, además de la elección de un elemento base sobre la que trabajar, tomar decisiones de diseño acerca del peso de cada parámetro en nuestro ejemplo de uso (que son el resultado de cada ronda, y el tiempo de duración de la misma) intentando a la par que fuera todo lo parametrizable posible, de cara a un hipotético uso en otros proyectos, sin que diera lugar a problemas estructurales de por si) y respecto a elementos externos que pueden influir tanto en la experiencia de juego como en el proceso de emparejamiento, como son las desconexiones intencionadas (o involuntarias) en medio de las partidas, tomando como referencia criterios utilizados por juegos como League of Legends para equilibrar el castigo a jugadores que intentan abusar del sistema, pero sin llegar a perjudicar en exceso a aquellos que puedan sufrir un percance técnico de manera puntual.

Al margen de esto, de cara a la elaboración de la memoria, he participado en la redacción, en menor o mayor medida, de prácticamente todos los capítulos de la misma, además de

darle formato e ir haciendo revisiones periódicas para solventar diferencias de estilo o pequeñas erratas.

Índice de figuras

Bibliografía

[1] K. Harkness, *Official chess rulebook*. D. McKay Co, 1970.

[2] M. E. Glickman, "Example of the glicko-2 system," *Boston University*, pp. 1–6, 2012.

[3] A. E. Elo, *The Rating of Chessplayers, Past and Present*. Arco, 1978.

[4] "ECF Grading Help Page." http://www.ecfgrading.org.uk/new/help.php#elo, 2016.

[5] J. Junyszek, "May playlist updates." https://www.halowaypoint.com/en-us/news/may-playlist-updates, 2018.

[6] H. Stenhouse, "CS:GO ranks, explained." https://www.pcgamer.com/csgo-ranks-explained/, 2021.

[7] C. Germain, "/dev: Updates on 2020 ranked matchmaking." https://na.leagueoflegends.com/en-us/news/dev/dev-updates-on-2020-ranked-matchmaking/, 2020.

[8] N. desconocido, "An in-depth look at the splatoon 2 ranking system." https://oatmealdome.me/blog/an-in-depth-look-at-the-splatoon-2-ranking-system/7D, 2018.

[9] J. O'Dell, "Finding the perfect match." https://www.guildwars2.com/en/news/finding-the-perfect-match/, 2014.

[10] Tetr.io, "Tetr.io." https://tetrio.team2xh.net/?t=faq, 2021.

[11] P. Showdown, "Pokemon showdown ladder help." https://pokemonshowdown.com/pages/ladderhelp, 2016.

[12] S. Saed, "Team fortress 2 patch overhauls matchmaking, revamps competitive mode." https://www.vg247.com/2018/03/29/team-fortress-2-patch-competitive-matchmaking-revamp/, 2018.

[13] Lichess, "Chess rating systems." https://lichess.org/page/rating-systems, 2021.

[14] M. E. Glickman, "The glicko system," *Boston University*, vol. 16, pp. 16–17, 1995.

[15] R. Herbrich, T. Minka, and T. Graepel, "Trueskill™: a bayesian skill rating system," in *Proceedings of the 19th international conference on neural information processing systems*, pp. 569–576, 2006.

[16] T. Minka, R. Cleven, and Y. Zaykov, "Trueskill 2: An improved bayesian skill rating system," *Tech. Rep.*, 2018.

[17] J. Menke, "Matchmaking for engagement: Lessons from halo 5." https://www.youtube.com/watch?v=0FoG4Jtpebs, 2020.

[18] M. Izquierdo, "Ranking systems: Elo, trueskill and your own." https://www.youtube.com/watch?v=VnOVLBbYlU0, 2017.

[19] G. Wallner, *Data Analytics Applications in Gaming and Entertainment*. CRC Press, 2019.

[20] J. Menke, "Skill, matchmaking, and ranking systems design." https://www.youtube.com/watch?v=-pglxege-gU, 2017.

[21] J. Van Dongen, "Why good matchmaking requires enormous player counts." http://joostdevblog.blogspot.com/2014/11/why-good-matchmaking-requires-enormous.html, 2014.

[22] J. Van Dongen, "Designing matchmaking for non-gigantic communities." http://joostdevblog.blogspot.com/2015/09/designing-matchmaking-for-smaller.html, 2015.

[23] J. Van Dongen, "The psychology of matchmaking." https://www.gamasutra.com/blogs/JoostVanDongen/20190308/338244/The_psychology_of_matchmaking.php, 2019.

Parte II

English

Capítulo 9

Introduction

9.1. Motivation

In the past few years, the format of online multiplayer video games has become very popular, and a lot of games have adapted in order to offer an online experience where users can measure their skills against other players. To this end, there have been a many different attempts at designing, improving or expanding matchmaking systems (which pair players) and classification systems (which classify players) both in traditional games [1, 2, 3, 4] and online multiplayer video games [5, 6, 7, 8, 9, 10, 11, 12].

One of the main challenges when looking to achieve a pleasant and fun competitive environment for users is to make matchmaking systems that allow players to face others with a similar level of skill, offering a challenge suitable for their own abilities.

Due to this, a lot of studies and approaches related to matchmaking in multiplayer video games have arisen (detailed in section 2 *Estado Del Arte*), specially those whose main concern is to offer a competitive experience. The vast majority of them are created under the premise that each player possesses a certain level of skill which is represented as a numerical value, or rating. Colloquially known as Elo, referencing the Elo system used to pair players in chess which was slowly introduced into video games as a simple way to pair online players. Slowly, new systems derived from Elo have surfaced that offer a more in-depth solution.

These systems have the main objective of establishing a way to measure players' aptitudes considering different factors (such as the result of a game or different behaviours or actions that might have taken place during the competition). These factors then allow the system to match those players whose skills are better in the scope of a certain video game or sport.

There are several design philosophies that offer multiple options when it comes to developing matchmaking systems, all of which are derived from the ones originally used in chess. Most of these extend the Elo system in order to obtain a more precise rating and and more precise pairings. The main issue is that these implementations are not easily accessible, either due to lack of open source code that is easy to reuse, or due to the complexity of this topic. Thus, our motivation is to create a system that can be easily

scaled and adapted to the needs of a specific game and can be adjusted as the developer needs.

9.2. Objectives

Our main objective is to design and develop a matchmaking system that can be implemented into an online video game. This system will work through a server with access to a user database and will use this information in order to make the pairings. Separately, it should be possible to update the rating of each of them according to their performance in previous games.

To design this system, we will study already existing systems that are already in use, so we can design ours from a stable foundation. We will also study different methods of rating players, as this is an important part of pairing them together.

We will also develop an online multiplayer video game to test this matchmaking system, as well as any other tests we might need to perform. This game will also be used in various tests with real players to determine its performance and quality.

As an additional goal, we also want to design this system in a way that allows it to be adapted and used in any kind of online videogame.

9.3. Work plan

For the development of this project we will internally, and with certain liberties, employ Scrum as a framework, dividing work into items that can be more easily distributed according to availability, capacity and knowledge.

To monitor progress we will have weekly meetings to adapt our tasks and establish new goals. We have already used this framework in multiple subjects throughout our career with satisfactory results.

We will organise meetings with our tutors in order to establish the different levels of priority for our pending tasks, as well as to ask questions and queries of a more technical nature.

Firstly we will investigate different matchmaking systems already used in other video games to use them as an example to follow when implementing our own. Afterwards, we will design the basic architecture of this matchmaking system, keeping in mind any restrictions that may need to be applied depending on its focus. Once designed, this architecture will be implemented, along with the connection between clients and server. Furthermore, we will test this architecture in search of error, simulating connections between clients and observing they proceed properly.

Finally, even though it is not the main focus of this project, we will need to develop a videogame as a use case, with the objective of using it to test the matchmaking system. Tests with real users will be carried out through this game, allowing them to play freely so as to study the system's performance and the data collected from these players.

9.4. Document structure

Chapter 2 *Estado Del Arte* details our investigation into matchmaking systems and the conclusions drawn from them, along with our decisions as to which systems use as an example and implement. The design of our system is explored in chapter 3 *Diseño del sistema de matchmaking*. Chapter 4 *Implementación* focuses on the implementation of this system, and the initial tests carried out with it.

The game design document for our game can be found in chapter 5 *Caso de estudio*, and the tests carried out with it are detailed in chapter 6 *Prueba con usuarios*.

URLs of the repositories used for this project:

- General repository, which contains the rest as submodules:
  - https://github.com/HoracioStudios/TFG
- Matchmaking Server:
  - https://github.com/HoracioStudios/Matchmaking-Server
- Rating update system:
  - https://github.com/HoracioStudios/Ranking-Update
- Game server controller:
  - https://github.com/HoracioStudios/ControlServidoresTeFeGe
- Game:
  - https://github.com/HoracioStudios/TeFeGe
- Dynamic library for connecting the game and the servers:
  - https://github.com/HoracioStudios/ClientCommunication
- Database connection:
  - https://github.com/HoracioStudios/MongoJS

Capítulo 10

Conclusions and Future Work

The main objective of this project has been to design and develop a working matchmaking system with a set of restrictions, which could then be implemented for an online video-game with these same restrictions. We believe we have achieved this: the system works through a server, independent from the actual video game, with access to a database where users' data is stored and can be utilised to answer matchmaking petitions, pairing online players in a balanced fashion, while also accounting for the time they may have spent searching for a match, broadening the search accordingly. In a separate server the users' ratings are updated according to their performance in-game.

For this, we have studied several matchmaking systems to learn how they're structured, and then studied different rating systems until we found the one that would be easiest to implement, adapt and expand: Glicko. However, this project is designed in such a way that developers can change this rating system at will, provided they edit the relevant code.

To put forth and test this project we have developed an online multiplayer video game where we could implement our matchmaking system, and it is this game that we have used to test the system with real users. A dynamic library that could handle connections between the matchmaking server and the client was also developed and, despite attempts to make it completely generic and applicable to any game or project, we haven't been able to fix the issue presented by how JSON objects are handled in C#.

In essence, this library is almost completely independent from our game and could theoretically be used in any other, but in the case that a developer wanted to change the information sent at the end of a match they would need to edit a single class from this library, the one that is then serialised into a JSON object to then send the match data to the database, due to restrictions imposed by C#.

Still, we offer a generic version of this library that only sends the minimum required match data (results and duration of each round, an opponent's id and the match's id), and this version can indeed be implemented into any game that meets the matchmaking system's requirements. We also offer inside the matchmaking server the option of redefining a function to allow the processing match data before it is sent to the database, in case this data needs to be analysed in some way.

10.1. Future Work

Although we are satisfied with how we have managed to complete the objectives set for this project, there still exist certain areas where the system can be improved or expanded.

Firstly, one of the more interesting challenges would be to include the option to pair users by teams. Although this is a process that could be done in a simplified fashion (first creating a team and then pairing those teams), this method would not take into account variables such as different degrees of skill, players choosing to play together, or many other factors that play into the proper formation of a team.

We suggest the inclusion of a set of predefined roles or playstyles which could then factor into the pairing system: many games allow players to define (or assign to the players automatically) their preferred playstyles, be it in terms of class (support, attack, defence) or playstyles (aggressive, defensive, stealthy, et cetera). These factors are then taken into account when pairing players in the same team so they complement eachother, or players with opposing playstyles so they compete.

Secondly, one also adjust the default parameters used to calculate ratings, which at the moment only include the results of a round or match. We have not included any other paarmeters so as to keep the system generic.

One could include new parameters like accuracy or the amount of damage dealt during a round, data which is already collected from the game. This data could be used to reflect a player's ability more accurately and, to tie into the last part, it could help generate player profiles that could factor into pairing a team with players that would complement eachother, making the game not just fair in terms of raw skill, but also when playing with unknown players. ANother parameter of interest would be the duration of a round or match, which could be used to reward players which manage to win in a quick fashion and to lower the punishment of a loss in the case that a player manages to survive for a long time.

When it comes to the more technical aspects, and thanks to the results obtained from carrying out tests with real users, we have learned that the computation of each round's result individually generates an important change in how ratings and deviations evolve over time. Similarly, we have also detected that the default values offered by Glicko could be altered, specially when it comed to deviations.

As we cannot reliably recalculate all ratings to factor in changed or new values, we must instead recommend carrying out more testing sessions to properly study these changes.

In the case that a reliable and active userbase can be established, another possible expansion would be to upgrade the system we currently use to its successor, Glicko-2, an expanded version that includes a parameter representing rating volatility. This system could be easily upgraded, as it already offers a way to generate new Glicko-2 ratings from regular Glicko ratings.

Lastly, we decided to design this system in a way such that it would be independent from our game. For this purpose we have designed a dynamic library which handles communication between client and server. However, when it comes to sending data to

the database, C# requires specifying a format for this data, while the servers and the database are designed to allow for flexibility in this format. We have developed a version of this library that only sends the minimum necessary data, and another that adapts to our game's needs. It is important to point out that this dynamic library is the only obstacle preventing this system from being completely generic when used with C#, and we have been able to test this genericity with languages that allow sending data with a flexible format.

Thus, it would be interesting to find a way to alter this dependency on the game and test that this matchmaking system can be applied to different games made in C# without the need to modify this library, while also offering the flexibility we wanted to offer from the get-go. This should take into account any game, be they similar to ours or of different genres, as long as they share the restriction of 1 vs. 1 matches. fin